

AD-A193 933

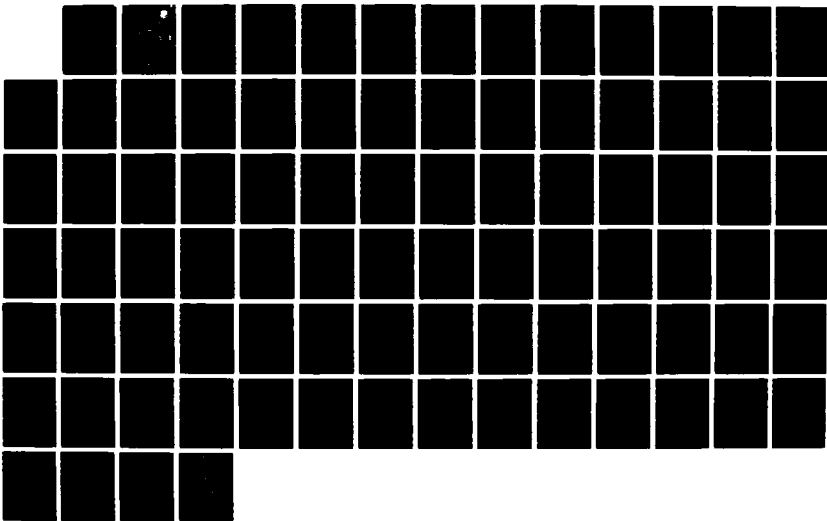
LOGLISP PROGRAMMING SYSTEM USERS MANUAL (U) HONEYWELL
SYSTEMS AND RESEARCH CENTER MINNEAPOLIS MN
J CARCIOFINI ET AL. NOV 87 RADC-TR-87-228

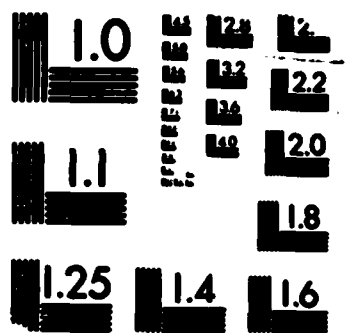
1/1

UNCLASSIFIED F38682-84-C-8121

F/G 12/5

NL





MICROCOPY RESOLUTION TEST CHART
 NBS 1963-A

AD-A193 933

AD-A193 933-228
Final Technical Report
November 1987



DTIC FILE COPY

LOGLISP PROGRAMMING SYSTEM USERS MANUAL

Honeywell Systems and Research Center

DTIC
ELECTE
APR 27 1988
S H D

James Carolofini, George Hadden, Timothy Colburn and Aaron Larson

APPROVED FOR PUBLIC RELEASE, DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffis Air Force Base, NY 13441-5700

88 4 26 145

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-87-249, Vol II (of two) has been reviewed and is approved for publication.

APPROVED:



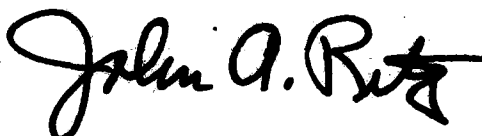
RICHARD M. EVANS
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COKE) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A			5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-228		
6a. NAME OF PERFORMING ORGANIZATION Honeywell Systems and Research Center		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)		
6c. ADDRESS (City, State, and ZIP Code) 3660 Technology Drive Minneapolis MN 55418			7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center		8b. OFFICE SYMBOL (If applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-84-C-0121		
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO 63728F	PROJECT NO 2532	TASK NO 01
					WORK UNIT ACCESSION NO. 08
11. TITLE (Include Security Classification) LOGLISP PROGRAMMING SYSTEM USERS MANUAL					
12. PERSONAL AUTHOR(S) James Carciofini, Timothy Colburn, George Hadden, Aaron Larson					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM Jun 85 TO Jun 87		14. DATE OF REPORT (Year, Month, Day) November 1987	
				15. PAGE COUNT 92	
16. SUPPLEMENTARY NOTATION N/A					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Logic Programming Programming Languages		
09	02		Loglisp Programming Environments		
			Logic Artificial Intelligence		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The Loglisp Programming System (LPS) is a Common Lisp environment extended to provide logic programming capabilities. The system consists of an interpreter, a compiler, an editor interface, and a user interface targeted to a VAX 11/780. This User Manual describes the features of the Loglisp language and the LPS system facilities that comprise overall environment. Enhancements to the Common Lisp environment are also detailed. The Computer Operations manual describes the installation of the LPS on the VAX 780 system detailing system requirements and procedures. <i>Keywords: logic programming</i>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL John J. Crowter			22b. TELEPHONE (Include Area Code) (315) 330-2973		22c. OFFICE SYMBOL RADC (COES)

DD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

UNCLASSIFIED

UNCLASSIFIED

Contents

Preface	v
1 Introduction To LogLisp	1
1.1 General Description	1
1.1.1 Lisp Environment	1
1.1.2 Logic Programming Capability	1
1.1.3 Interface of Lisp and Logic	2
1.1.4 LogLisp Programming Environment	2
1.2 Logic Programming in Lisp	3
1.2.1 Unification	3
1.2.2 Deduction Trees	6
1.2.3 Lisp Reduction	8
2 How To Use The LogLisp Interpreter	11
2.1 Beginning a LogLisp Session	11
2.2 Individual Clause Assertion	13
2.3 Defining Whole Procedures	14
2.4 Executing a LogLisp Program	15
2.4.1 The LogLisp Query Function - <code>lps:query</code>	15
2.4.2 The LogLisp Query Macros	17
2.5 Special Resolution Forms	19
2.5.1 Quitting The Overall Query	19
2.5.2 Failing Or Succeeding A Goal	20
2.5.3 Suspending A Node	22
2.5.4 Other Special Resolution Forms	24
2.6 Overflowing The Node Stack	26
2.7 Indexing Clauses for Efficiency	26
2.7.1 The Value of Indexing	26



Accession For	
20 TIS GRA&I	<input checked="" type="checkbox"/>
22 TIC TAB	<input type="checkbox"/>
24 Unannounced	<input type="checkbox"/>
Justification	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

2.7.2	Specifying and Finding Indexing Keys	27
2.7.3	Setting a Path Specifier for a Procedure	29
2.8	Modifying the Knowledge Base	29
2.8.1	Modifying Clauses within Procedures	29
2.8.2	Deleting Whole Procedures	30
2.9	Viewing the Knowledge Base	30
2.10	Reduction	31
2.10.1	Atoms	32
2.10.2	Applicative Forms	32
2.10.3	Macros	33
2.10.4	Common Lisp Special Forms	34
2.10.5	Other Reduction Special Forms	37
3	The LogLisp Tracing Facility	39
4	How To Use The LogLisp Compiler	43
4.1	What the LogLisp Compiler Does	43
4.2	Calling the Compiler	43
4.3	Mode Declarations	44
4.3.1	The Purpose of Mode Declarations	44
4.3.2	The Formal Syntax and Meaning of Mode Declara- tions	45
4.3.3	Using Mode Declarations	46
5	The LogLisp User Interface	49
5.1	VaxLisp Extensions	50
5.1.1	Input Editor	50
5.1.2	The LogLisp Command Processor	54
5.1.3	Extended Function Documentation Facility—defun+	56
5.2	Emacs Extensions	59
Appendix A - LogLisp Programming System Computer Operations Manual		

Preface

This Users Manual for the LogLisp Programming System (LPS), RADC Contract F30602-84-C-0121, is to provide the user's non-ADP personnel with the information necessary to effectively use the system.¹

Documents applicable to the history and development of the LogLisp Programming System project are specified in section 1.2 of the LPS Functional Description and in section 1.2 of the LPS Test Plan. The system/subsystem specifications for the LPS interpreter and compiler might also be of interest. Also, in order to effectively develop LogLisp programs the user will need documentation for Emacs.

1. Beane, J., Carciofini, J. and Colburn, T., "LogLisp Programming System Functional Description". Honeywell Systems and Research Center, April 12, 1985.
2. Beane, J., Carciofini, J., Colburn, T. and Lukat, R., "LogLisp Programming System Test Plan", Honeywell Systems and Research Center, August 1, 1985.
3. Carciofini, J., Colburn, T. and Lukat, R., "LogLisp Programming System Interpreter System/Subsystem Specification", Honeywell Systems and Research Center, December 16, 1985.
4. Carciofini, J., Colburn, T. and Lukat, R., "LogLisp Programming System Compiler System/Subsystem Specification", Honeywell Systems and Research Center, April 15, 1986.

¹This preface covers Section 1 (General) of the documentation specification for users manuals given in DoD-STD-7935. This specification was written more with general data processing applications in mind; the LPS, on the other hand, is an artificial intelligence program development tool. Therefore, Honeywell (the developer) has elected not to follow the remainder of this specification. Instead, the presentation is organized as indicated in the table of contents.

5. Stallman, R.. *GNU Emacs Manual*. fifth edition. October, 1986, copyright 1985, 1986 by Richard M. Stallman.

The general nature of programs developed for the LogLisp Programming System can be classified as artificial intelligence application development tools. As such, they will be used for intelligent database management as well as expert system development.

The sponsor of the LogLisp Programming System project is Rome Air Development Center at Griffiss Air Force Base in New York. The developer is Honeywell Incorporated's Systems and Research Center in Minnesota. Both Rome Air Development Center and Honeywell will use the completed system.

Technical terms, abbreviations and acronyms unique to this project are defined in this manual where they are introduced.

Chapter 1

Introduction To LogLisp

1.1 General Description

The LogLisp Programming System (LPS) described here is a Common Lisp programming environment extended to provide logic programming capabilities. Although the primary target machine is a VAX running VAXLisp under VAX/VMS, the LPS may be ported to any machine supporting full Common Lisp, for example Symbolics or Sun workstations.¹

1.1.1 Lisp Environment

Since the LPS is built on top of Lisp, the full power of Common Lisp is available, and indeed the LPS may be used only for its Lisp capability if the user so desires. The Common Lisp environment includes tools for error handling, debugging, pretty printing, Lisp compiling, I/O and file editing (using Gnu Emacs).

1.1.2 Logic Programming Capability

The LPS's logic programming capability, implemented in a language called Logic, enables a declarative style of programming based on predicate calculus representation and Horn clause resolution. This style is similar to that of Prolog, but it is more general in that the user can influence the direction of the deduction cycle by specifying how the deduction tree is to be searched, and by making calls to Lisp. The logic programming capability includes

¹ The LPS user interface is dependent on VAXLisp and thus can only be run on the VAX. On other machines, however, the LPS is fully accessible through direct function calling.

mechanisms for asserting facts in a knowledge base and deducing answers to queries of the knowledge base. The LPS may be used as a pure logic programming system, with no calls to Lisp from Logic, if the user so desires.

1.1.3 Interface of Lisp and Logic

The LPS provides an interface between Lisp and Logic in the following two ways:

1. Since the Logic primitives (queries) of the LPS return Lisp objects, they (the Lisp objects) may be used as arguments to standard Lisp functions. In particular, since the Logic primitives are themselves invoked as Lisp functions, they may be embedded in other Lisp functions which manipulate the results.
2. The goals of a LogLisp procedure clause can be interpreted for their applicative (Lisp) meaning instead of as declarative (Logic) goals. If a goal predicate symbol has a Lisp meaning, either as a functional form, special form, or macro, the programmer can arrange for the goal to be reduced according to the reduction semantics for Common Lisp. Similarly, selected arguments, or indeed arguments of arguments, of goals may have reduction applied to them. Thus a LogLisp programmer may arrange for the reduction of Lisp forms at any level of his or her logic program. A consequence of reduction is that Logic queries can recursively invoke Logic queries, since a logic query is itself a Lisp function call, and can appear anywhere in a LogLisp clause body.

1.1.4 LogLisp Programming Environment

The LogLisp programming environment is an extension of the Common Lisp programming environment on the VAX (or other machine running Common Lisp) which features the familiar interactive style of Lisp program development, along with a powerful Emacs style editor. LogLisp programming is also interactive, with logic procedures typically asserted, tested, edited, saved and compiled at will, just as Lisp functions are similarly defined, tested, edited, saved and compiled. A powerful and flexible LogLisp user interface is also provided, including a LogLisp command processor, input editor and Lisp extensions for Emacs.

1.2 Logic Programming in Lisp

This section provides a tutorial style introduction to the concepts of logic programming and Lisp reduction, as general background for the instructional sections to follow. We begin with a description of the primary logic programming inferencing tool (**unification**), continue with the primary logic programming data structure (**deduction trees**), and conclude with a description of the primary mechanism by which logic and Lisp programming in LogLisp interact (**reduction**).

1.2.1 Unification

A **logic program** is a collection of facts, or declarations. A logic program is so-called because the model for representing its facts is **predicate logic**, or **predicate calculus**. For example, the predicate logic representation of the unconditional fact "Fred is older than Otto" is

older-than(fred, otto)

and can easily be represented in Lisp as the S-expression

(older-than fred otto)

We call *older-than* the **predicate** and *fred* and *otto* its **arguments**. A formula associating a predicate and some arguments, as in (older-than fred otto), is called a **predication**. In LogLisp, facts will always be represented as *lists* of predications. Thus to be correct the fact "Fred is older than Otto" is really the list containing the above predication:

((older-than fred otto))

This representation is necessary since facts may be conditional. The predicate logic representation of the conditional fact "So-and-so is older than such-and-such if so-and-so was born before such-and-such" is

$(\forall x)(\forall y)[\text{born-before}(x, y) \rightarrow \text{older-than}(x, y)]$

This formula is read, "for all *x* and for all *y*, if *x* was born before *y* then *x* is older than *y*". It also can be rendered in Lisp, as

((older-than ?x ?y) (born-before ?x ?y))

where it is understood that (1) `?x` and `?y` represent universally quantified variables (that is, they occur as objects of the "for all..." specification) and (2) the head of such an S-expression represents the consequent part of the declaration and the tail its antecedents, or **goals**. As it stands, the meaning of such an expression is unambiguous, but for clarity we will include the `<-` character to indicate the direction of the implication and also the `&` character to separate the occurrences of multiple goals:

```
((older-than ?x ?y) <- (born-before ?x ?y))
```

`?x` and `?y` are examples of **logic variables**. In predicate logic, variables are identified by their being objects of the quantifiers \forall and \exists . In a logic programming system logic variables are identified lexically. We stipulate that logic variables are atoms whose print-names begin with a `?` character.

A fact, whether conditional or unconditional, in a logic program is also called a **clause**. Clauses with the same predicate constitute a **logic procedure**. The collection of logic procedures making up a logic program is also called a **knowledge base**. The object of logic program execution is the substitution of terms for logic variables in a given predication, or **input query**, which makes that predication deducible from the knowledge base.

The process of logic program execution is a series of attempts at matching the structure of an input query against the structure of some head predication of some clause in the knowledge base. (This process is also sometimes referred to as **backward chaining**.) The original input query is a predication provided by the user of the knowledge base. Suppose the knowledge base is composed of our clauses

```
((older-than fred otto))
((older-than ?x ?y) <- (born-before ?x ?y))
```

and the input query is

```
((older-than fred otto)).
```

Since there are no logic variables in this query, the posing of it is in essence a question whether Fred is older than Otto. Since the query `(older-than fred otto)` is matched, verbatim, against the first clause the answer to the query is "yes". Thus in this case logic program execution is just a matter of finding a duplicate copy of the query in the knowledge base.

In order for two non-variables to match, they must be of the same type. Thus the atom `fred` and the string `"fred"` will not match. Numbers are

tested using the Common Lisp `=` predicate, strings use the Common Lisp `string-equal` predicate, and all other atoms are tested using `eql`. Conses are tested recursively on their cars and cdrs.

But the power of logic programming comes from the ability to also match on the logic variable. Suppose the input query is

```
((older-than fred ?z))
```

This differs in that it includes the logic variable `?z`. Initially this variable does not stand for anything: it is **uninstantiated**. Thus it makes no sense to interpret this query as asking whether Fred is older than `z`. Rather, it is a request for an enumeration of those objects which when substituted for `z` make the statement "Fred is older than `z`" true. Since "Fred is older than Otto" is a fact in the knowledge base, the input query will elicit the response "Otto". Note that if "Fred is older than Butch" is also a fact in the knowledge base,

```
((older-than fred butch)).
```

then the answer to our query is "both Otto and Butch".

In this case not only must execution of the logic program do pattern matching on a list structure, but it must also take into account the presence of a logic variable. Although there is not a verbatim match between the query `(older-than fred ?z)` and the clause `(older-than fred otto)` as they stand, there is if `otto` is substituted for `?z` in the query. This type of pattern matching lies at the heart of logic programming and is called **unification**. When an input query is selected for unification, we say that the logic procedure associated with that query's predicate is **called**. Logic program execution is a series of unification steps, or logic procedure calls.

Sometimes logic program execution requires the unification process to remember a logic variable binding and use this binding in a subsequent unification attempt. Suppose that there is no explicit statement in our knowledge base of Fred being older than anyone. Instead, this time the knowledge base contains only the clauses

```
((older-than ?x ?y) <- (born-before ?x ?y))  
((born-before fred otto))
```

and the input query is still

```
((older-than fred ?z))
```

The response to the query will again be "otto", but this time the action of the logic machinery is more complicated. Let us call this machinery the **inference engine**.

The inference engine will attempt to match the query against the head of the clause (older-than ?x ?y) <- (born-before ?x ?y) and it will succeed, binding fred to the logic variable ?x and binding the logic variable ?z to the logic variable ?y. These bindings are temporary, however, because they are contingent on the antecedent goal being satisfied, namely (born-before ?x ?y). Thus the matching cycle repeats, with the goal (born-before ?x ?y). Again, we say that this goal calls the logic procedure for born-before. This differs from the initial input query in that the logic variables ?x and ?y have been instantiated. Thus the unification process will find a successful match between this new query and the clause (born-before fred otto) because (1) ?x was previously bound to fred, and (2) ?y was previously bound to ?z which is as yet unbound and therefore matches successfully with otto under the substitution otto for ?z.

1.2.2 Deduction Trees

Thus it is seen that the data area manipulated through logic program execution is, in general, a growing set of logic variable bindings, also called an **environment**. The successful unification of an input query against a knowledge base clause head produces a (possibly null) set of logic variable bindings which extends the current environment.

Sometimes it happens that an input query matches against more than one clause head. For example, suppose we have the following clauses in the knowledge base:

```
((older-than ?x ?y) <- (born-before ?x ?y))
((older-than ?x ?y) <- (born-before ?y ?x))
((born-before fred otto))
```

Now the input query

```
((older-than fred ?z))
```

will match the heads of the first two clauses, producing separate and independent sets of bindings, each extending the original environment in isolation from the other. In other words, the environment **branches** into more than one set of variable bindings, each becoming an environment unto itself but sharing the variable bindings in force before the branch. In the

abstract. a logic program's main data area is a **deduction tree** of logic variable bindings, embodying a separate environment for each node of the tree at any point during the execution. The process of determining the descendants of a given node in this tree is called **resolution**, because it uses a rule of inference of the same name.

There are choices to be made in the concrete implementation of this idea. The most important is to decide how this deduction tree is to be traversed, that is, in what order are its nodes to be visited. Think of a deduction tree node as being a list of goals remaining to be proved in a query and the current environment. The logic programming language Prolog goes by the policy that, once a goal successfully unifies with a clause head, work begins immediately on the next goal of the node's goal list, rather than attempting to unify the same input query with alternate clauses. Thus the tree of variable bindings is constructed in a strict depth-first fashion, and continues that way until unification fails or an answer to the query is found. Upon failure all bindings made since the last point when alternative clauses could be matched are thrown away and matching is attempted on the next clause (called **backtracking**). Thus only a single branch (environment) of the abstract tree is in effect at any time.

Another control policy is to match all possible clauses against an input query before calling the goals of any of these clauses. This implementation more closely corresponds to the abstract deduction tree described above. At any point during program execution a number of environments are in effect. Although only one environment is current in the sense that it is the one currently being extended, there are a number of other live ones waiting their turn for extension. Depending upon the criteria by which a waiting leaf is selected for resolution, the deduction tree can grow in a more or less balanced manner. The more balanced a tree grows, the more jumping there is between environments (called **context switching**).

LogLisp offers the option of traversing the deduction tree in either of these two modes, strict depth-first with backtracking or breadth-first with no backtracking, as well as in any combination of them. Combinations of these modes are effected through a limited backtracking method which does a Prolog style depth-first search unless the programmer requests a suspension. A suspension consists of taking the current node and placing it on a waiting heap according to a solution cost heuristic. (In going to the heap, this node joins other nodes which were current at some time and then suspended. If processing is strictly depth-first, no node will ever be put on the heap.) Then, the node from this waiting heap with the cheapest

solution cost is selected, it is reinstated as the current node in the deduction tree, and processing continues. The solution cost heuristic can be as simple as a constant weight, or it can be computed using a complicated formula. Suspension can also be conditional.

Besides being able to suspend a deduction, the programmer can also explicitly fail a node according to any criterion he/she chooses. Section 2.5 describes how to control the deduction through programmed suspension and failure.

1.2.3 Lisp Reduction

Reduction is the LogLisp mechanism by which logic programs can use the power of Lisp. For example, suppose in our procedure for *older-than* we want to include the fact that "*a* is older than *b* if *a*'s age is greater than *b*'s age". The meaning of this fact includes the concept of numerical comparison, the evaluation of which can be performed by a Lisp system function. In LogLisp we can construct the following clause to embody this fact about *older-than*:

```
((older-than ?a ?b) <- (age ?a ?n1) & (age ?b ?n2) & !(> ?n1 ?n2))
```

This states that *a* is older than *b* if the age of *a* is *n1*, the age of *b* is *n2*, and *n1* is greater than *n2*. Notice that the last goal is prefixed with an exclamation point (!). This is a LogLisp reader macro which expands the last goal into

```
(reduce-term (> ?n1 ?n2))
```

The *reduce-term* form tells LogLisp to treat its argument as Lisp would and evaluate it. This is not exactly the same as calling the Lisp form *eval* on (> ?n1 ?n2), since logic variables have to be treated specially, but it is similar. Thus if processing has reached the last goal of the above clause and it evaluates, the value is interpreted as an indication of success or failure. If the value is *nil* the goal has failed, otherwise it has succeeded.

A call to Lisp need not always appear as a clause goal in itself, but may be embedded within a goal as in

```
(age ?c !(+ ?n1 20))
```

Here the goal is a logic predication (read "the age of *c* is the sum of *n1* and 20"), which does not have a Lisp meaning. However, if ?n1 is instantiated to an expression that has a value then LogLisp will be able to recognize that

the goal's last argument is evaluable. The interpreter will rewrite the goal with the appropriate substitutions. For example, if ?n1 is instantiated to 5 then the goal will be rewritten to

```
(age ?c 25).
```

In this case the goal has not been evaluated but it has been reduced, by evaluating one of its sub-forms.

Reduction deals with predications which are syntactically the same as S-expressions. However, there are some basic semantic differences due to LogLisp's use of Lisp atomic forms. As arguments to logic procedure calls, Lisp atoms are used for two purposes: logic constants and logic variables. Logic constants are their own values. For example, the value of the symbol *fred* is *fred*. One mentions a logic constant in a logic predication just by using it (unquoted), as in

```
(older-than fred otto)
```

In Lisp, however, when one wants an atom rather than its value, the atom is quoted as in

```
(eq 'fred (car '(fred is tall)))
```

because Lisp evaluates an unquoted *fred* by returning its lisp binding, which may be something other than *fred* altogether.

Thus if a LogLisp programmer wants to use a symbol in a non-logic way within a LogLisp clause or query, for example, to *setq* a symbol to a (Lisp) binding and later to get that binding, he must anticipate that the forms within which that symbol is going to be used will be subject to reduction, which assumes that an unquoted atom is a self-evaluating logic constant. In order to access that atom's binding, *eval* must explicitly be called on it. For example, if the Lisp binding of the symbol *tall-list* is the list

```
(otto fred bill)
```

then in order to access this list for membership one must write

```
!(member ?x (eval tall-list))
```

instead of just

```
!(member ?x tall-list)
```

This is a frequent source of errors in LogLisp programs which call out to Lisp. For more on this aspect of the Logic Lisp interface, see section 2.5.2.

Note also that any Lisp atom used as a logic variable, like `?x` above, either has a logic binding at the time reduction is called on it or it does not. If it does, the binding, and not the variable itself, is given to the Lisp reduction machinery. Finding this binding is called **dereferencing** the variable. Thus if `?x` is bound to the atom `fred` then the form

```
!(member 'fred (eval tall-list))
```

will be given to the reduction machinery at runtime, even though the source goal is

```
!(member ?x (eval tall-list))
```

If the variable does not have a logic binding, or it dereferences (possibly recursively) to another variable, then the (last) variable is given to the reduction machinery, where it is interpreted as having no Lisp value and reducing to itself. Thus no applicative form in which it appears will have a Lisp value either, and such a form will be sent back, reduced but unevaluated, to the inference machinery of LogLisp. In our example, reduction will return

```
(member ?x '(fred otto bill))
```

to Logic, and if no logic procedure for `member` has been defined, this procedure call will fail.

Chapter 2

How To Use The LogLisp Interpreter

2.1 Beginning a LogLisp Session

LogLisp is installed at VAX/VMS sites in the form of a suspended Lisp image extended with the functions necessary for LogLisp execution. The command for invoking the suspended LPS image is `lps`. In order to make this command available, and to properly initialize VAX Lisp and Gnu Emacs, three files are essential in your `SYS$LOGIN:` directory. All can be created using the VMS `CREATE` command.

LOGIN.COM This command procedure file is automatically executed each time you log in. It must contain the following lines:

```
$ lispexten.init [memsize]  
$ gnu.emacs.init [logout]
```

memsize is an optional parameter which tells VAX Lisp how much working memory to use. The default is 10000 pages of dynamic space. *logout* can be indicated to tell whether you want the system to inform you of any processes (particularly Gnu Emacs processes) which are still running when you attempt to logout.

LISPEXTEN-INIT.LISP This is an initialization file which is loaded into the VAX Lisp environment each time you enter the LPS. It must contain the function call

```
(cp:enable-command-processor)
```

Executing this function is necessary for using the LPS command interface.

.EMACS This is an initialization file required for GNU Emacs. It must contain the function calls

```
(local-customizations)
```

```
(use-flow-control)
```

The first function makes available the extended Emacs commands for Lisp mode. The second function is optional but should be used when your host-to-terminal communication protocol uses control-S and control-Q for flow control. Since Gnu Emacs does not follow this protocol, undesirable behavior can occur when these control characters are received by the host. The `(use-flow-control)` function essentially tells Gnu to ignore control-S and control-Q and to replace them with control-\ and control-], respectively.

Once this is done, the suspended image is invoked simply by typing

```
$ lps
```

The dollar sign (\$) is the Digital Command Language prompt. When the suspended image has been loaded you will receive the prompt

```
Ed Lisp CP>
```

The "Ed" and "CP" designations indicate that the input line editor and command processor are available as part of the Lisp extensions provided.

All of the LogLisp function, macro and special form symbols are external to the `lps` package. That is, you can reference a symbol by prefixing its name with `lps:`, as in `lps:query`. However, you can make these symbols available without the prefix if you "use" the LPS package. This is done by typing the form:

```
(use-package 'lps)
```

to Common Lisp. However, the shorter symbols remain available only as long as you stay in the package which uses `lps`. In this manual we will always refer to symbols with their prefixed names, with the understanding

that the prefix may be dropped when the `lps` package is used. (See Guy Steele's book, *Common Lisp, The Language*, for a complete discussion of packages.)

The LPS is now ready to use as documented in the following sections.

2.2 Individual Clause Assertion

The simplest way to enter clauses into your knowledge base is with the `lps:assert-clause` function:

```
(lps:assert-clause clause)
```

This function installs *clause* in the knowledge base. The syntax for a *clause* is:

```
(head [<-|<--|if] [goal {[*|and] goal}*])
```

where *head* is itself a list of the form

```
(predicate {argument}* . [NIL|variable])
```

(Notational conventions for describing forms are also given in Steele.) *predicate* must be a symbol, but not a logic variable, while *argument* can be any Common Lisp object whatsoever. The argument list constituted by *argument** can be dotted so long as the dotted tail is a variable. *goal* can be any Common Lisp object except that if *predicate* is one of the special resolution or reduction forms in section 2.5 or 2.10 it must follow that form's syntax. Also, a goal's argument list may not be dotted.

If a clause does not conform to this syntax, an informative message is given, a fatal error is signalled and the VaxLisp debugger is entered. The `lps:assert-clause` function is frequently called when talking directly to Lisp. If so, the user would in this case exit the debugger, fix the line using the input line editor (see section 5.1.1), and re-enter it. Following are two examples of the use of `lps:assert-clause`.

```
(lps:assert-clause '((foo a b)))
(lps:assert-clause '((foo ?x ?y) if (bar ?x ?z) &
                        (baz ?z ?y)))
```

If a clause is asserted to a compiled procedure, the procedure must be recompiled for the object code to reflect the change.

2.3 Defining Whole Procedures

We now describe a method for defining whole procedures using just one form. The `lps:defpredicate` macro allows you to define whole logic procedures at once.

```
(lps:defpredicate name
  [:mode mode-template]
  [:index path]
  clause*)
```

where *name* is a symbol naming a predicate, *path* is a valid path specifier (see section 2.7 on indexing), *mode-template* is a valid mode declaration template (used only in conjunction with the LogLisp compiler—see section 4.3.3) and *clause* is a clause conforming to the syntax given in Section 2.2, and whose head predicate is the same as *name*.

If an `lps:defpredicate` form does not conform to the syntax given above, an informative message is given, a fatal error is signalled and the VaxLisp debugger is entered. The `lps:defpredicate` macro is frequently called when working in an editor and evaluating Lisp forms out of it. (See section 5.2.) If so, the user would in this case exit the debugger, fix the form using the editor, and re-evaluate it.

The `lps:defpredicate` form arranges to define the LogLisp procedure *name* with clauses asserted under the index path specifier given by the `:index` specification. If no path specifier is explicitly given, it defaults to `(car)`. If clauses under the predicate *name* are already defined, they are removed and replaced by the new clause assertions. The exception is where the evaluation of the `lps:defpredicate` form encounters a syntax error, in which case the original clauses are left undisturbed. Following is an example of how to use `lps:defpredicate` instead of `lps:assert-clause` in the previous section:

```
(lps:defpredicate foo
  :index (car)
  ((foo a b))
  ((foo b c))
  ((foo ?x ?y) (bar ?x ?z) (baz ?z ?y)))
```


2.4 Executing a LogLisp Program

Executing a LogLisp program is equivalent to querying a knowledge base. There is one primitive function for querying the knowledge base. Three macros also exist that use this primitive query function. First we describe the primitive Logic function `lps:query`. Then we present the three query macros `lps:all`, `lps:any` and `lps:one`. Usually `lps:query` is used under Lisp program control while the macros are used under Logic program control to make recursive calls on Logic. This is because the macros make sure that any logic variables in their goals are instantiated. The macros are also more convenient than certain typical patterns of use of the `lps:query` function.

2.4.1 The LogLisp Query Function - `lps:query`

The primary construct for querying a knowledge base is the `lps:query` function. The syntax of a call to `lps:query` is

```
(lps:query goal-list
          &key
          :template
          :solution-limit
          :ignore-duplicates)
```

where the parameters are described as follows:

goal-list This is a list of initial goals to be proved. It must conform to the syntax of a clause body, that is, it is like the `cdr` of a clause. A *goal-list* of `((foo ?x))` will cause LogLisp to search for a substitution for `?x` such that `(foo ?x)` is true. A *goal-list* of `((foo ?x) (bar ?x ?y))` will cause LogLisp to search for substitutions for `?x` and `?y` such that both `(foo ?x)` and `(bar ?x ?y)` are true.

:template This is a form typically containing logic variables which is returned as an answer template whenever a solution is found. For example, the template consisting simply of `'?x` indicates that the form of the answer is to be the binding of `?x` when the solution is found. The quote is necessary in the template since `lps:query` is a Lisp function and therefore evaluates all of its arguments. The template `'(?x ?y)` indicates that the form of the answer is to be the list containing the bindings of `?x` and `?y`.

Note that templates can be reduced. Thus the template `'!(?x ?y)` will cause the reduction mechanism to check if `?x` is bound to a symbol with a functional definition. If so, and `?y` is bound to something that has a value, then the answer will be the result of calling that function with that value as an argument.

The default answer template is an association list of the bound variables appearing in the goal list, where the key is the variable name and the value is the variable's binding.

Since a query might succeed more than once, `lps:query` returns the list of answers found during the deduction. If no answers are found, `lps:query` returns `NIL`. Note that when no variables appear in *goal-list* and the default template (an a-list) is used, then if *goal-list* succeeds `NIL` will be the answer, but `lps:query` returns the list of answers. So if there is one success, `(NIL)` is returned; if there are two successes `(NIL NIL)` is returned; etc.

:solution-limit (default: `:infinity`) This is either an integer indicating the number of solutions requested, or the keyword `:infinity` indicating that all solutions (within the limits of the deduction tree search) are to be found.

:ignore-duplicates (default: `nil`) This parameter indicates whether duplicate solutions are to be ignored in the construction of the query's answer list. If it is `nil`, all solutions will be collected in the answer list regardless of duplicates. If the parameter is `t`, a solution will be ignored if it is (Common Lisp) equal to any other already found. If it is a function, it must be an equality predicate to be used for determining whether the solution is a duplicate. Mathematically speaking, if this parameter is not `nil`, the answer list will be like a set; if it is `nil`, the answer list will possibly be a bag.

Now we give some simple examples. Suppose the knowledge base contains the following predicates:

```
(lps:defpredicate foo
  ((foo a b))
  ((foo a c))
  ((foo ?u ?v) (bar ?v ?u)))
```

```
(lps:definepredicate bar
  ((bar b a)))
```

Then we might have the following queries and answers:

```
Query: (query '((foo a ?x)))
Answer: (((?x . b)) ((?x . c)) ((?x b)))

Query: (query '((foo a ?x)) :template '?x)
Answer: (b c b)

Query: (query '((foo a ?x)) :template '?x
                          :ignore-duplicates t)
Answer: (b c)

Query: (query '((foo a ?x)) :template '?x
                          :solution-limit 1)
Answer: (b)

Query: (query '((foo a ?x)) :template '(cons 57 ?x)
                          :solution-limit 1)
Answer: ((cons 57 b))

Query: (query '((foo a ?x)) :template '!(cons 57 ?x)
                          :solution-limit 1)
Answer: ((57 . b))

Query: (query '((foo ?x b) (foo ?x c)) :template '?x
                          :ignore-duplicates t)
Answer: (a)
```

2.4.2 The LogLisp Query Macros

It is an error to use the `:solution-limit` keyword option with the query macros. The arguments to these macros are subject to variable instantiation when they appear in the body of a clause, and thus are more convenient to use than the `lps:query` function. Defaults for the `:template` and `:ignore-duplicates` keyword options are the same as for `lps:query`. Descriptions and examples of the query macros follow. Note that goals need not be placed in lists and arguments need not be quoted. The examples assume the same knowledge base as before.

`(lps:all goal* keyword-option-pairs*)`

The `lps:all` macro is similar to the `lps:query` function with the `:solution-limit` keyword set to `:infinity`. Examples:

Query: `(lps:all (foo ?x b) (foo ?x c) :template ?x)`

Answer: `(a a)`

Query: `(lps:all (foo ?x b) (foo ?x c) :template ?x
:ignore-duplicates t)`

Answer: `(a)`

`(lps:any k goal* keyword-option-pairs*)`

The `lps:any` macro is similar to the `lps:query` function with the `:solution-limit` keyword set to `k`. Examples:

Query: `(lps:any 3 (foo a ?x) :template ?x)`

Answer: `(b c b)`

Query: `(lps:any 3 (foo a ?x) :template ?x
:ignore-duplicates t)`

Answer: `(b c)`

Query: `(lps:any 1 (foo a ?x) :template ?x)`

Answer: `(b)`

`(lps:one goal* keyword-option-pairs*)`

The `lps:one` macro is similar to the `lps:any` macro called with `k = 1`. However, rather than returning a list with one answer, `lps:one` just returns the answer. Examples:

Query: `(one (foo a ?x) :template ?x)`

Answer: `b`

Query: `(one (foo a z))`

Answer: `:NO-SOLUTIONS-FOUND`

Note that if `lps:one` is called with a goal for which there are no solutions, the atom `:NO-SOLUTIONS-FOUND` is returned rather than `NIL`. This is to distinguish it from the case, mentioned earlier, in which a deduction succeeds but returns `NIL` because no variables appear in the goal and the default template was used.

2.5 Special Resolution Forms

The LogLisp `lps:query` function and macros provide a certain amount of control over the deduction process by allowing the programmer to set a solution limit. But this is very high-level control over the set of solutions generated and does not offer any control within the generation of any one particular solution. LogLisp offers this control by providing special forms for quitting the overall query, failing a particular branch of the deduction tree, and suspending a node for possible resumption at a later time. There are also three special forms which simulate logical connectives, and one for constructing goals at run-time. All of these "special resolution" forms are only interpreted as such when they appear as goals in clauses.

2.5.1 Quitting The Overall Query

There can be many reasons for quitting a query while it is in progress. For example, there may be an infinite number of ways to get a solution, in which case the query will not stop if left unconstrained. As mentioned, one way to stop such a query is to use the `:solution-limit` keyword when invoking Logic. But there may be other criteria for stopping an unconstrained search. For example, the programmer might wish for the search to stop after a certain amount of time has elapsed, or after a certain number of deduction tree nodes have been tried. Or more realistically, there might be a specialized criterion specific to the application for stopping the query. For these purposes, LogLisp provides a special form which when it occurs as a goal causes the current deduction to stop and return the list of answers generated so far. If the current deduction is nested in another, the higher level deduction continues.

```
(lps:quit)
```

This causes the deduction to be stopped and the current answer list returned. Normally, `lps:quit` will be invoked conditionally, using the `lps:logic-if` form (see section 2.5.4). For example, suppose the procedure `foo` is defined as follows:

```
(lps:defpredicate foo
  ((foo ?x ?y) (lps:logic-if !(quit-check ?x)
                             (lps:quit)
                             (bar ?y))))
((foo ?x ?y) (baz ?x) (boo ?y)))
```

Suppose also the current goal is (foo zip zap). Then if quit-check is a Lisp function which when called with zip as an argument returns a non-NIL value, the overall query is stopped: the second clause for foo is not tried. Otherwise, control passes to the goal (bar zap).

2.5.2 Failing Or Succeeding A Goal

Forms are provided for explicitly initiating backtracking and succeeding goals.

```
(lps:fail)
```

When used as a goal in a clause, this form unconditionally fails and backtracks in the deduction tree to the last point where alternative clauses have yet to be tried. If there is no such point, lps:fail will have the same effect as lps:quit. A similar form is provided for unconditionally succeeding a goal:

```
(lps:succeed)
```

In most cases it is more useful to be able to fail or succeed conditionally. For example, suppose the predicate foo is defined as follows:

```
(lps:defpredicate foo
  ((foo ?x ?y) (lps:logic-if !(fail-check ?x)
                             (lps:fail)
                             (lps:succeed)))
  ((foo ?x ?y) (baz ?x) (boo ?y)))
```

Suppose also the current goal is (foo zip zap). Then if fail-check is a Lisp function which when called with zip as an argument returns a non-NIL value, backtracking commences and the next clause is tried. Otherwise, the goal succeeds and processing continues with whatever goal is next, if any. The lps:succeed form is for most purposes a no-op.

For a more realistic example of explicit search control, consider the following predicate:

```
(lps:defpredicate older-than
  ((older-than fred otto))
  ((older-than otto spike))
  ((older-than spike butch))
  ((older-than ?x ?y) (older-than ?x ?z) (older-than ?z ?y)))
```

Since the last clause always succeeds, is recursive, and introduces two new goals, there will be depth-first runaway and the query

```
(query '((older-than fred ?who)))
```

will not terminate. To stop it, one can notice that the query will never require that the deduction tree go deeper than four levels, one for the initial query and one for each of the intermediate steps involved in proving that Fred is older than Butch. Thus if `depth` were a Lisp function which returned the current depth in the deduction tree, and the last clause is rewritten

```
((older-than ?x ?y) (lps:logic-if !(> (depth) 3)
                                   (lps:fail)
                                   (lps:logic-and
                                    (older ?x ?z)
                                    (older ?z ?y))))
```

then the depth-first runaway would be cut off and both correct answers returned. (See section 2.5.4 for a description of `lps:logic-and`.)

Note that in order for this to work the user must have initialized some Lisp variable to zero and defined the function `depth` in such a way as to increment this variable and return its value.

As a matter of style, the use of reduction to access Lisp state from within Logic should be avoided. A better example of how to program the `older-than` procedure is the following:

```
(lps:defpredicate older-than
  ((older-than fred otto))
  ((older-than otto spike))
  ((older-than spike butch)))

(defpredicate older
  ((older ?x ?y) (older-than ?x ?y))
  ((older ?x ?y) (older-than ?x ?z) (older ?z ?y)))
```

This method does not cause depth-first runaway, and does so in a way which does not require a depth check. It also requires only about one quarter the time of the other method.

2.5.3 Suspending A Node

The feature which distinguishes LogLisp the most from Prolog is that which allows the suspending of a node. Unlike failing, which effectively aborts a deduction tree branch, suspending gathers up the state of the deduction into a *waiting node*, saves it on a queue called the *waiting heap*, and backtracks from the current point. Waiting nodes are inserted into the heap according to a solution cost, or heuristic which estimates the cost of finding a solution from the current point. If and when all nodes have been thus suspended, the waiting node with the least solution cost is selected and installed as the root of a new deduction tree.

The form for suspending deduction is:

```
(lps:suspend cost)
```

where *cost* is a form which evaluates to a number representing the solution cost. It is an error for *cost* to evaluate to a non-number.

As an example of using the `lps:suspend` form, consider a logic program for finding a path between two nodes in a directed graph. The graph can be represented by a *connected* predicate defined as follows:

```
(defpredicate connected
  ((connected a b))
  ((connected b c))
  ((connected c d))
  ((connected d e))
  ((connected a f))
  ((connected f e)))
```

Paths between any two nodes on this graph can be computed by the procedure `path` along with its subsidiary procedures:

```
(defpredicate path
  ((path ?x ?y ?z) if
    (pathrecurs ?x ?y ?z (?x ?y))))

(defpredicate pathrecurs
  ((pathrecurs ?s ?e (?s ?e) ?ex) if
    (connected ?s ?e))
  ((pathrecurs ?s ?e (?s . ?t) ?ex) if
    (connected ?s ?n) and
```



```
(not-elif ?n ?ex) and
(pathrecurs ?n ?e ?t (?n . ?ex))))
```

```
(defpredicate elif
  ((elif ?x (?x . ?y)))
  ((elif ?x (?h . ?t)) if (elif ?x ?t)))
```

```
(defpredicate not-elif
  ((not-elif ?x ?l) if !(not (any 1 (elif ?x ?l)))))
```

The procedure `pathrecurs` will attempt to compute the path as a list beginning with `?s` and ending with `?e` subject to the restriction that the path cannot include nodes from `?ex` (the object being to avoid getting caught by cycles in the graph). The procedures `elif` and `not-elif` are the Logic implementations of list membership and non-membership, respectively. Note the use of the recursive call to Logic in `not-elif`.

The query

```
(all (path a e ?path) :template ?path)
```

will return the list of paths

```
((a b c d e) (a f e)).
```

which is the correct answer. But if what is desired is the *shortest* path, this procedure is inefficient since the longer path is computed first. We can guarantee that the shortest path is found first by keeping track of the length of the path computed so far and suspending the current deduction tree node according to this length. Thus `path` and `pathrecurs` can be replaced with `path-smart` and `pathrecurs-smart`:

```
(defpredicate path-smart
  ((path-smart ?x ?y ?z) if
    (pathrecurs-smart ?x ?y ?z (?x ?y) 0)))
```

```
(defpredicate pathrecurs-smart
  ((pathrecurs-smart ?s ?e (?s ?e) ?ex ?length) if
    (connected ?s ?e))
  ((pathrecurs-smart ?s ?e (?s . ?t) ?ex ?length) if
    (connected ?s ?n) and
    (not-elif ?n ?ex) and
```

```
(= ?new-length !(1+ ?length)) and
(suspend ?new-length) and
(pathrecurs-smart ?n ?e ?t (?n . ?ex) ?new-length)))
```

```
(defpredicate ==
  ((= ?x ?x)))
```

The procedure `==` simply unifies its arguments. If neither of the arguments is a variable they must be equal for `==` to succeed. If one of them is a variable it will be bound to the other. If both are variables then the first will be bound to the second.

Now the query

```
(all (path-smart a e ?path) :template ?path)
```

will return the list of paths

```
((a f e) (a b c d e)),
```

and the list `(a f e)` will have been found first. To make sure that it is *all* that is found, the query

```
(one (path-smart a e ?path) :template ?path)
```

will return the single answer

```
(a f e),
```

and no further processing will be done.

2.5.4 Other Special Resolution Forms

LogLisp provides four more forms which have special meaning as predicates. The first three are somewhat akin to the logical connectives *and*, *or* and *if...then...else*. To distinguish them from their Common Lisp counterparts, they are called `lps:logic-and`, `lps:logic-or` and `lps:logic-if`.

```
(lps:logic-and conjunct*)
```

When this form appears as a goal it succeeds only if each of the *conjuncts* succeeds. Thus, for example, the clause

```
((bachelor ?x) (lps:logic-and (unmarried ?x) (male ?x)))
```

is equivalent to

```
((bachelor ?x) (unmarried ?x) (male ?x))
```

If there are no *conjuncts*, `lps:logic-and` automatically succeeds. The `lps:logic-and` form is useful when more than one goal are involved in a branch of the `lps:logic-if` form (see below).

```
(lps:logic-or disjunct*)
```

This goal succeeds only if some *disjunct* does. Thus, for example, the clause

```
((organic ?x) (lps:logic-or (animal ?x) (plant ?x)))
```

is equivalent to the pair of clauses

```
((organic ?x) (animal ?x))  
((organic ?x) (plant ?x))
```

If there are no *disjuncts*, `lps:logic-or` automatically fails.

```
(lps:logic-if test then [else])
```

When this form appears as a goal, an attempt is made to prove *test*. If successful, *then* becomes the new goal and the `lps:logic-if` form succeeds only if *then* does. Otherwise, if *all* attempts to prove *test* fail, *else* becomes the new goal and the `lps:logic-if` form succeeds only if *else* does. If *test* fails and there is no *else* alternative, the `lps:logic-if` form automatically fails.

The last special resolution form allows the run-time construction of goals.

```
(lps:call expression)
```

expression must be something which, when instantiated, can be interpreted as a goal. For example, if *?x* is bound to `(foo bar)` then `(lps:call ?x)` will invoke the goal `(foo bar)`. But *expression* need not be a variable; it can be a structured object which contains a variable. Suppose the source goal is `(lps:call (?y 13))`. Then if *?y* is bound at run-time to *baz*, the invoked goal will be `(baz 13)`. If *expression* does not instantiate to a valid goal, an error is signalled, just as though the goal had failed to parse at clause input time.

2.6 Overflowing The Node Stack

Although much of LogLisp's run-time memory management is handled by Lisp, the primary run-time data structure, called the *node stack*, is not. The node stack is a preallocated array used to represent the state of a Logic deduction in much the same way that the control stack is used by Lisp to represent function calling. If the node stack is used up before a query is finished a continuable error will be signalled:

Node stack overflow.

If continued: deduction will resume with a larger stack.

2.7 Indexing Clauses for Efficiency

2.7.1 The Value of Indexing

A clause indexing facility is provided for run-time efficiency. The most fundamental level of indexing is at the predicate level. That is, given a query

```
(lps:query '((foo ?x)))
```

unification will only be attempted against clauses in the logic procedure *foo*. But indexing is also desirable within logic procedures themselves. For example, suppose the following clauses are asserted for the procedure *older-than*:

```
(defpredicate older-than
  ((older-than fred otto))
  ((older-than otto spike))
  ((older-than spike butch))
```

Suppose also that this group of clauses is not indexed in any way. Then if we pose the query

```
(query '((older-than spike ?who)))
```

three unifications will be necessary to obtain the correct response

```
((?who . butch)))
```

To eliminate unnecessary unification attempts, LogLisp by default will organize the clauses of the procedure in a hash table which is keyed on the atoms **fred**, **otto** and **spike**, that is, the first arguments of the clause heads. The value of each table entry then is the (list containing the) clause for which the key is the clause head's first argument. At run-time, the first argument of the goal, in this case **spike**, will be recognized as the key, and the hash table value for **spike**, namely the list containing the clause **((older-than spike butch))**, will be retrieved. Thus only the clause with a chance to unify will be tried, rather than all three.

It is sometimes desirable to index procedures on some element other than the first argument. For example, suppose the procedure for **older-than** is as above, but the query instead is

```
(query '((older-than ?who butch)))
```

Then despite indexing the procedure on the first argument, all three clauses of the procedure will have to be tried, since the first argument of the goal is a variable, and a variable can unify with anything. In this case it is more desirable for the procedure to be indexed on the second argument, for again only the one clause with a chance of unifying will be retrieved.

2.7.2 Specifying and Finding Indexing Keys

Indexing keys are specified through constructs called **path specifiers**. A path specifier is a (possibly null) list of symbols each of which is either **car** or **cdr**. This list specifies a path into a clause head's argument list, ending at the desired indexing key. Following a path specifier proceeds as follows. The first **car** or **cdr** accessor function in the path specifier is applied to the argument list, producing a value. The next **car** or **cdr** accessor function in the path specifier is then applied to this value, producing a new value. This process continues until each of the functions in the path specifier has been applied to the result of the previous application. The last value is the indexing key.

Lisp users will recognize these path specifiers as performing a task similar to the **car** and **cdr** composition functions such as **cadr**. The difference is that path specifiers are followed naturally from left to right, while **car** and **cdr** composition functions perform the applications from right to left. For example, the path specifier **(cdr car)** is in some sense equivalent to **cadr**. But composition functions are not used, both because they can only go up

<i>Clause Head</i>	<i>Path Specifier</i>	<i>Index Key</i>
(foo a b)	(car)	a
(foo a b)	(cdr car)	b
(foo a b)	(car car)	undefined
(foo (a c) b)	(car car)	a
(foo (a c) b)	(car cdr)	(c)
(foo (a c) b)	(car cdr car)	c
(foo a . b)	(cdr)	b
(foo a b c d)	nil	(a b c d)
(foo a b ?x)	(cdr cdr car)	?x

Table 2.1: Index Keys

to four levels of application, and because it is necessary to do list bounds checking to prevent run-time errors.

Table 2.1 gives examples of clause heads, path specifiers for those clause heads, and the resulting index key. Remember that the path specifier is applied to the clause head's argument list, and not the whole clause head itself. There are a number of remarks to be made about the use of path specifiers.

1. This method allows indexing on keys which are inside of **structured arguments**. This is particularly useful when it is desired that clauses be indexed on functors or functor arguments.
2. It is possible that the application of a path specifier to an argument list is undefined. that is, the path of car's and cdr's cannot be completed. LogLisp makes provisions for this situation, such that the input/output behavior of a program is not affected. However, the performance of the program may be degraded, due to the fact that clauses for which the path specifier is undefined must be handled in a general way.
3. Index keys need not be logical constants. They may also be Lisp atoms which are not symbols, for example, arrays or structures. Clauses which are keyed on such objects are also stored in the hash table. Other non-constant objects are logical variables and conses. Clauses which are keyed on objects such as this are not stored in the hash table. They are simply kept on lists, one for those clauses keyed on variables, the other for those keyed on conses.

4. The indexing method handles dotted structures in the clause head just as any other structure.

2.7.3 Setting a Path Specifier for a Procedure

The default path specifier for every procedure is (*car*). To use anything else one must define the procedure using `lps:defpredicate` with the `:index` keyword. For example, the form

```
(lps:defpredicate tall
  :index (car cdr car)
  ((tall (father a)))
  ((tall (father b)))
  ((tall (father c)))
  ((tall (father d)))
  ((tall (father e))))
```

will optimally index its clauses.

2.8 Modifying the Knowledge Base

2.8.1 Modifying Clauses within Procedures

The `lps:defpredicate` form can be used in conjunction with an emacs-style editor in Lisp mode, so that the programmer may Lisp-evaluate complex `lps:defpredicate` forms from within the editor, and modify them as required (see section 5.2.) When a changed `lps:defpredicate` form is re-evaluated, the effect is to delete any changed clauses and assert new ones in their place.

One may also surgically remove (`retract`) a clause from its procedure's structure. This is done through the `lps:retract-clause` function, whose syntax is

```
(lps:retract-clause clause)
```

where *clause* is the clause to be retracted, verbatim. For example,

```
(lps:retract-clause '((older-than spike butch)))
```

removes the clause `((older-than spike butch))` from the `older-than` procedure. If *clause* is not in the knowledge base, `lps:retract-clause` returns `NIL`, otherwise `T` is returned. If *clause* happens to be the only one in the

procedure, the procedure will still exist but it will be empty. If a clause is retracted from a compiled procedure, the procedure must be recompiled for the object code to reflect the change.

2.8.2 Deleting Whole Procedures

While `lps:assert-clause` can and `lps:defpredicate` does add new whole procedures to the knowledge base, procedures can also be deleted using the `lps:delete-predicate` form:

```
(lps:delete-predicate predicate-specifier)
```

where *predicate-specifier* is either

- A symbol naming the predicate of the procedure to be deleted,
- A list of symbols naming predicates to be deleted, or
- The keyword `:all`, meaning that the entire knowledge base is to be cleared

`lps:delete-predicate` not only removes all the clauses from the procedure; the procedure itself is deleted from the knowledge base.

2.9 Viewing the Knowledge Base

If the user routinely asserts clauses through `lps:defpredicate`, then the method for viewing his knowledge base is simply to view the file in which the `lps:defpredicate` forms reside. But if the knowledge base was not defined with `lps:defpredicate` forms, there are two options for viewing the contents of procedures.

First, the contents of a particular procedure (or procedures) may be viewed through the `lps:pprint-predicate` function, which pretty-prints a form for the procedure at the user's terminal, or, optionally, on some other designated stream. The syntax for `lps:pprint-predicate` is

```
(lps:pprint-predicate predicate-specifier &optional stream)
```

where *predicate-specifier* is as for `lps:delete-predicate` and *stream* is an optional i/o stream, which defaults to the value of the Common Lisp system variable `*terminal-io*`.

Second, an actual `lps:defpredicate` form (not just a format string) for a procedure can be returned using the `lps:get-predicate` function:

`(lps:get-predicate predicate)`

where *predicate* is a symbol naming the predicate for which an `lps:def-predicate` form is desired.

If all the user wants to see is the names of the predicates currently in the knowledge base, the function call

`(lps:list-all-predicates)`

returns a list of them.

There is also a function which returns the clauses of a predicate as a list. The form of this function is:

`(lps:get-predicate-clauses predicate)`

where *predicate* is a symbol naming the predicate.

Finally, the function

`(lps:list-all-clauses)`

returns a list of all the clauses in the knowledge base.

2.10 Reduction

Reduction is the method by which LogLisp programs invoke the power of Lisp. In the following sections we will define how the LogLisp interpreter reduces an expression in terms of its reduction and value. There are three rules which govern the reduction of an expression:

1. Only arguments to the `lps:reduce-term` form are reduced.
2. An expression must be reduced as far as possible.
3. If an expression has a value, it must be equivalent to the value of the reduction of the expression.

The syntax of the `lps:reduce-term` form is

`(lps:reduce-term expression)`

It is an error if `lps:reduce-term` does not have exactly one argument. As a matter of convenience, the reader macro `'` can be used instead of `lps:-reduce-me`. Thus, `(lps:reduce-term expression)` is abbreviated by

!expression

Rule 2 above is to ensure that, for example, the logic goal

```
(foo (+ (- 10 3) 17) bar)
```

is reduced all the way to

```
(foo 24 bar)
```

and not merely to

```
(foo (+ 7 17) bar).
```

The third rule will be explained in the following sections.

The classes of expressions for which reduction is defined are atoms, applicative forms, macros, Common Lisp special forms and reduction special forms.

2.10.1 Atoms

An atom in Common Lisp is any Lisp object which is not a cons. The value of an atom is the atom itself except for logic variables which have no value. All atoms are reduced, that is, an atom's reduction is the atom itself. Note again that in order to obtain the binding of a symbol *a* this scheme requires that `(eval a)` be written rather than just *a* as in Lisp.

a has value *a* and reduction *a*

36 has value 36 and reduction 36

"Hello" has value "Hello" and reduction "Hello"

?x has no value and reduction ?x

2.10.2 Applicative Forms

Let us call a **proper identifier** any symbol that is not a logic variable. An applicative form is an expression of the form

$$(f\ e_1\ e_2\ \dots\ e_n)$$

where *f* is a proper identifier that does not name a special form or macro. Such an expression has a value if *f* names a function and all of the arguments e_1, e_2, \dots, e_n have values. If the expression does not have a value (that is, either *f* does not name a function or at least one of e_1, e_2, \dots, e_n does not

have a value) then its reduction is $(f\ e'_1\ e'_2\ \dots\ e'_n)$ where e'_i is the reduction of e_i . Otherwise let v_1, v_2, \dots, v_n be the values of e_1, e_2, \dots, e_n respectively and let r be the result obtained by applying f to the values v_1, v_2, \dots, v_n . Then the value of the expression is r and the reduction is `(quote r)`

To understand the need to quote r recall that an expression and its reduction must have the same value. Now if an expression evaluates to an atom there is no problem, because in that case its reduction is an atom and we have already seen that an atom which is not a logic variable has itself as a value. In general, however, if an expression has a value it need not be atomic. For example, the value of `(list a b c)` is `(a b c)`. But we do not want to say that the reduction is also `(a b c)`, because this form is non-atomic and can be interpreted as an applicative form itself with value, say, `frobboz`. Thus we would have a situation in which a form, namely `(list a b c)`, and its reduction, namely `(a b c)`, would have values which are not necessarily equivalent, namely `(a b c)` and `frobboz`, respectively.

We can guarantee that a form and its reduction have the same value by quoting the reduction and formulating reduction semantics for the special form `(quote k)` which are intuitive plus give the results we seek. According to these semantics, the value of `(quote k)` is k and its reduction is itself. Thus in our example we have that the reduction of the form `(list a b c)` is `(quote (a b c))`, and we find that the value of the form is the same as the value of its reduction, namely `(a b c)`.

```
(+ 4 2) has value 6 and reduction 6
(+ ?x (+ 4 2)) has no value and reduction (+ ?x 6)
(f (+ x (+ 4 2))) has no value and reduction (f (+ ?x 6))
(list a b c) has value (a b c) and reduction '(a b c)
```

2.10.3 Macros

The value and reduction of a macro call are simply the value and reduction of the expansion of the macro call. In most cases macros can be called in LogLisp clauses with no problem. However, caution is required when using macros which expand into forms which contain Common Lisp special forms having no useful LogLisp reduction semantics (see section 2.10.4). For example, although `cond` is a Common Lisp macro, it is implemented in VAX Lisp using the special form `let`. Since there are no special LogLisp reduction semantics for `let`, using `cond` in a clause will not generally produce the desired behavior. Care should be taken so that, if a macro expands to a

block	if	progv
catch	labels	quote
compiler-let	let	return-from
declare	let*	setq
eval-when	macrolet	tagbody
flet	multiple-value-call	the
function	multiple-value-prog1	throw
go	progn	unwind-protect

Table 2.2: Common Lisp Special Forms

form containing Common Lisp special forms, they are limited to `if`, `progn`, `quote` and `setq`.

2.10.4 Common Lisp Special Forms

A special form is an expression of the form $(f\ e_1\ e_2\ \dots\ e_n)$ where the arguments are not necessarily evaluated and, although they always return a value, they are used primarily for control and for their side effects. The special forms of Common Lisp are listed in Table 2.2.

As far as contributing to the Logic-Lisp interface, we desire only the following capabilities over and above the normal applicative Lisp evaluation. We need to (1) conditionally evaluate Lisp forms, (2) serially evaluate Lisp forms, (3) quote Lisp forms, and (4) assign new values to Lisp symbols. For these purposes, we need to define special reduction semantics for just four of these special forms, respectively: `if`, `progn`, `quote` and `setq`. Many other common forms like `cond`, `and` and `or` are implemented as macros in Common Lisp that expand into these special forms. Thus these forms will also be available to the LogLisp programmer, subject to the provisions mentioned in section 2.10.3.

The rest of the special forms can not easily be given meaningful reduction semantics. If the LogLisp reduction mechanism encounters any of them it will be considered to be reduced and have no value. Thus using it as a goal or part of a goal is generally not useful, unless the LogLisp programmer has given the special form name a Logic meaning. These forms can not be given meaningful reduction semantics either because they do not make sense in the context of Logic, or because of implementation problems. Most of these forms have a body part which is evaluated in a locally modified Lisp environment. The forms which fit this description are listed in Table 2.3.

block	labels	macrolet
catch	let	progv
compiler-let	let*	unwind-protect
flet		

Table 2.3: Special Environment Forms

For example, *let* introduces bindings during the evaluation of its body. Thus for it to have meaningful reduction semantics a modified Lisp environment must be simulated during the reduction of the body. This would not be impossible to implement, but (1) the demand for this sort of capability is slight, and (2) it would be computationally expensive.

Following we give both the Common Lisp semantics and the LogLisp reduction semantics for the four Common Lisp special forms which have LogLisp meaning.

if test then [else]

Common Lisp Semantics This special form corresponds to the *if-then-else* construct found in most procedural programming languages. First the *test* form is evaluated. If the result is non *nil*, the *then* form is evaluated and its result returned. If the result is *nil*, the *else* form is evaluated and its result returned. In the latter case, if the *else* form is missing, *nil* is returned.

LogLisp Reduction Semantics First the *test* form is reduced. If the value of the reduction is non *nil* then the *then* form is reduced and its reduction result (i.e., its reduction-value pair) is returned. If the value of the reduction of the *test* form is *nil* then the optional *else* form is reduced and its reduction result is returned. If there is no *else* form then *nil* is returned as the value and reduction. In the case where the *test* form does not reduce to a value then the *if* form has no value and its reduction is obtained by replacing the *test* form with its reduction in the original *if* form.

*progn form**

Common Lisp Semantics This special form evaluates the forms, in order from left to right. The values of all the forms but the last are discarded; the

value of the last form is returned by `progn`.

LogLisp Reduction Semantics Each form is reduced, in order from left to right. If all the forms have values then the reduction result of the last form is returned. If a form *f* with no value is found, then the `progn` form has no value and its reduction is a `progn` form with the reduction of *f* as its first form and the (unreduced) forms to the right of *f* as its remaining forms.

`quote object`

Common Lisp Semantics This special form simply returns *object*, which is not evaluated.

LogLisp Reduction Semantics The form is reduced and has value *object*, which is not subject to instantiation. That is, any logic variables appearing in *object* will not be recognized as such.

`setq {var form}*`

Common Lisp Semantics This special form is the simple variable assignment statement of Lisp. The forms are evaluated and the results stored in the variables, one at a time, from left to right. The variable names are not evaluated. `setq` returns the last value assigned.

LogLisp Reduction Semantics Each pair `{var form}` is handled one at a time from left to right. It is an error for *var* to be anything but a symbol or a logic variable. If *var* is a symbol and the reduction of *form* yields a value, then the value is bound (in the Lisp sense) to *var*. If a pair `{var-i form-i}` is found such that either *var-i* is a logic variable or *form-i* does not yield a value, then the `setq` form has no value and its reduction is a `setq` form whose first pair consists of *var-i* and the reduction of *form-i*, and whose remaining pairs are the unreduced pairs to the right of `{var-i form-i}` in the original `setq` form. Otherwise, every pair `{var-i form-i}` results in a Lisp binding and the reduction result of the last *form* is returned as the reduction result of the `setq` form.

Remaining Special Forms

For the remaining Common Lisp special forms we define no special LogLisp reduction semantics. That is, if encountered by the reduction machinery these forms are assumed to be fully reduced and have no value.

2.10.5 Other Reduction Special Forms

There are six other reduction special forms that have no Common Lisp counterparts but nevertheless are handled specially by the LogLisp reduction machinery. They can only be used in clause bodies.

`(lps:logic object)`

Intuitively, `(lps:logic object)` specifies that the result of the evaluation of *object* is to be interpreted as a logic expression rather than a Lisp object. The most obvious effect of this is to suppress the quoting of non-atomic values.

If *object* has the value *v* then the value and reduction of `(lps:logic object)` is the value and reduction of *v*. Some examples:

- Suppose *a* has no function definition. Then `(lps:logic (list a b c))` has no value and reduction `(a b c)`
- `(lps:logic (list + 1 2))` has value 3 and reduction 3
- `(lps:logic (list cons s r))` has value `(s . r)` and reduction `'(s . r)`

Notice that the first example illustrates how the `lps:logic` form can be used to suppress the quoting of non-atomic values.

When *object* has no value the expression `(lps:logic object)` has no value and reduces to `(lps:logic r)` where *r* is the reduction of *r*. For example, if *a* does not name a function, the expression `(lps:logic (a b c))` has no value and reduces to `(lps:logic (a b c))`.

`(lps:irred object)`

This form has reduction *object* and no value. It is typically used for suppressing reduction of a form that occurs within the scope of `lps:reduce-term`.

`(lps:quasi-quote object)`

The expression `(lps:quasi-quote object)` is reduced and has value *object*. The difference between `lps:quasi-quote` and `quote` is that any variables in *object* are instantiated by `lps:quasi-quote` but not by `quote`.

`(lps:ground object)`

The `lps:ground` form is similar to `lps:quasi-quote` except that `(lps:-ground object)` has the value *object* only if *object* contains no unbound logic variables. Otherwise the form has no value. As with `lps:quasi-quote`, `(lps:ground object)` is reduced.

`(lps:logic-ground object)`

The form `(lps:logic-ground object)` is equivalent to `(lps:logic (lps:-ground object))`.

`(lps:variable object)`

This special form can be used to determine whether or not a logic variable is bound at runtime. If *object* is an unbound logic variable then the form `(lps:variable object)` has value and reduction `t`. Otherwise the form has value and reduction `nil`.

Chapter 3

The LogLisp Tracing Facility

The two macros `lps:trace-logic` and `lps:untrace-logic` control the trace facility. The macro `lps:trace-logic` allows the user to specify particular predicates to be traced or that all predicates should be traced. The syntax of the `lps:trace-logic` macro is:

```
(lps:trace-logic [:all | predicate*])
```

When the `:all` keyword is used all predicates are traced, otherwise only the predicates specified will be traced. If all predicates are being traced, this will continue to be true even if new predicates are subsequently defined. If no arguments are given to `lps:trace-logic` it returns a list of the predicates currently being traced or the keyword `:all` if all predicates are being traced.

The `lps:untrace-logic` macro can be used to turn tracing off for specific predicates or for all predicates. The syntax of this macro is:

```
(lps:untrace-logic [:all | predicate*])
```

When the `:all` keyword is used all tracing for all predicates is turned off, otherwise only the predicates specified will stop being traced. If no arguments are given the function turns off all tracing.

The tracer displays information about the flow thru invocations of logic procedures. A logic procedure consists of a group of clauses all with the same predicate name in their head. As the system searches for the solution to a query various events are displayed for those predicates that are being traced. These events are as follows:

Call This event indicates a new invocation of a logic procedure. Thus a unique integer is assigned to this invocation and all future events

involving this invocation will reference this number. This number is called the invocation identifier.

Exit This signals the exit of a logic procedure. When a logic procedure has succeeded this event is signaled. In other words some clause in the logic procedure has been successfully used by resolution.

Redo When a goal has failed and backtracking ensues, logic procedures are reentered to see if there is some other way to satisfy their goals. This event is signaled for each such procedure as it is backtracked into.

Fail This event is signaled when a logic procedure fails. This happens when resolution has exhausted all ways of using the logic procedure clauses to satisfy the goal that caused the invocation. An invocation can fail without ever exiting, or after exiting and redoing several times.

Quit When the resolution special form `lps:quit` is processed the quit event is signaled for all of the currently active invocations of the current query.

For all invocations there will be only one call event and either one fail event or one quit event. But there may be many exit and redo events for the same invocation.

The following example will be used to explain the tracer output in more detail.

```
Ed Lisp CP> (lps:defpredicate true-list
              ((true-list nil))
              ((true-list (?h . ?t))))

TRUE-LIST
Ed Lisp CP> (lps:trace-logic :all)
All predicates are now being traced.
Ed Lisp CP> (lps:all (true-list (a b)) :template yes)
>1 Call: (TRUE-LIST (A B))
>.2 Call: (TRUE-LIST (B))
>..3 Call: (TRUE-LIST NIL)
>..3 Exit: (TRUE-LIST NIL)
>.2 Exit: (TRUE-LIST (B))
>1 Exit: (TRUE-LIST (A B))
>* Solution found: NIL
>* Looking for another solution
```

(YES)

Each line of the tracer output begins with one or more > characters. The number of these characters indicates the nesting level of queries. In other words it shows how many recursive calls to the query function are active. Next a series of dots is printed to provide an indentation pattern that indicates which invocations are within others. Then for normal events the invocation number appears followed by the event name and the instantiated goal. There are several special events that are not associated with a particular invocation. These begin with a * character and are followed by a description of the event.

Chapter 4

How To Use The LogLisp Compiler

4.1 What the LogLisp Compiler Does

The LogLisp compiler translates source LogLisp procedures into code which executes faster than the LogLisp interpreter can run. The compiler accomplishes this by taking into account the types of head and goal arguments, by unwinding much of the recursion involved in structured terms, and eliminating much of the run-time memory management overhead incurred by the interpreter. Compiled LogLisp object code is modeled on the abstract Prolog instruction set created by D. H. D. Warren, extended with additional instructions expediting reduction.

4.2 Calling the Compiler

The smallest unit of input to the compiler is a procedure. Typically, clauses are asserted, using `lps:assert-clause` or `lps:defpredicate`, and then the procedure is compiled using the `lps:compile-predicate` function. The syntax is:

`(lps:compile-predicate predicate-specifier)`

where *predicate-specifier* is either

- A symbol naming the procedure to be compiled.
- A list of symbols naming procedures to be compiled, or

- The keyword `:all`, meaning that the entire knowledge base is to be compiled

If an indicated predicate is not a symbol, or it does not have a LogLisp procedure definition, an error is signalled. The user may return a procedure to interpreted form simply through the `lps:uncompile-predicate` function:

```
(lps:uncompile-predicate predicate-specifier)
```

where again *predicate-specifier* is as before.

For compiling files, the ordinary Common Lisp `compile-file` function can be used on files containing `lps:defpredicate` forms.

4.3 Mode Declarations

4.3.1 The Purpose of Mode Declarations

The purpose and benefits of mode declarations are best explained using the example of concatenating two lists:

```
(lps:defpredicate concat
  ((concat nil ?x ?x))
  ((concat (?a . ?x) ?y (?a . ?z)) (concat ?x ?y ?z)))
```

Declaratively, this procedure simply states what must be true for entities to stand in the relationship of `concat`. Procedurally, it shows either (1) how to concatenate two lists, or (2) how to construct a list such that it and another, when concatenated, result in a third. To restrict the procedural semantics of `concat` to either (1) or (2) requires input from the user of `concat`. There is no way that LogLisp can selectively apply restricted procedural semantics without this input. This input is given in the form of *mode declarations*.

If the user intends to use `concat` to always accept two list structures and create a third list structure that is the concatenation of the first two (procedure (1) above), the compiler can use this information to more efficiently classify variables within structures and also to eliminate type checking code. Thus the user may declare that the first two arguments to `concat` are to be regarded as input (read-only) arguments, and the last is to be output (write-only). In this way the user guarantees to the compiler that the first two arguments to `concat` will never be variables and the third will always be a variable. A mode declaration thus always applies to a predicate's argument list. In the present case the mode declaration for `concat` would thus be

(READ READ WRITE)

This form is called the *mode template* for concat.

4.3.2 The Formal Syntax and Meaning of Mode Declarations

A mode declaration is either structured or unstructured. An unstructured mode declaration is any of the symbols

- READ
- WRITE, or
- UNDECLARED

A structured mode declaration is a cons tree structure whose leaves are unstructured mode declarations (except that the cdr of the last cons may be NIL). Following are some valid mode declarations:

- (READ)
- (READ (UNDECLARED WRITE))
- WRITE
- (READ . UNDECLARED)

The meaning of mode declarations is given recursively as follows:

WRITE The corresponding procedure argument or argument element is guaranteed to be or dereference to an unbound variable.

READ The corresponding procedure argument or argument element is guaranteed to not be an unbound variable and, if it is a cons, to not have an unbound variable in its scope. That is, the argument is fully instantiated.

UNDECLARED The corresponding procedure argument or argument element might be anything at all.

$(md_1 \dots md_n)$ The corresponding procedure argument or argument element is guaranteed to be a cons, whose car and cdr are guaranteed according to the mode declarations md_1, \dots, md_n . For example.

(READ) guarantees the argument or argument element will be a cons whose car is a non-variable and whose cdr is NIL.

(UNDECLARED (WRITE)) guarantees the argument or argument element will be a cons whose car is undeclared and whose cdr is a cons whose car is a cons whose car is a variable

(WRITE . UNDECLARED) guarantees the argument or argument element will be a cons whose car is a variable, and whose cdr is anything at all.

4.3.3 Using Mode Declarations

Typically, a user declares modes for a procedure when evaluating an `lps:-defpredicate` form (see Section 2.3). Like an indexing path, a mode declaration for a procedure is keyworded, only one mode declaration is permitted, and it must appear before any clauses. If no mode declaration is given, the default is UNDECLARED. The mode declaration given above for `concat` would therefore be specified within a `lps:defpredicate` as follows:

```
(lps:defpredicate concat
  :mode (read read write)
  ((concat nil ?x ?x))
  ((concat (?a . ?x) ?y (?a . ?z)) (concat ?x ?y ?z)))
```

Note that in general a clause's "argument list" is usually a cons, but it may also be an atom (suppose the clause head is `(foo . ?x)`). Where a procedure's mode template does not match the structure of its argument list, the following rules apply:

1. If the template is not structured and has mode *mode*, that is, *mode* is READ, WRITE or UNDECLARED, but the argument list is a cons, the car and cdr of the argument list are given mode *mode* also.
2. If the template and argument list are both structured, but the argument list is "longer" than the template (that is, there are more top-level elements in the argument list than in the template), the "extra" elements are given a mode of UNDECLARED.
3. If the template and argument list are both structured, but the template is "longer" than the argument list (that is, there are more top-level elements in the template than in the argument list) the "extra" modes are discarded.

4. If the template is structured and its car or cdr is also structured, the above rules also apply between the embedded structure and the corresponding argument list element.

Usually, a mode declaration is given for a procedure using the `lps:def-predicate` form as shown above. Occasionally, however, a user may wish to change the mode declaration of a procedure which has already been defined. This is accomplished with the `lps:set-mode-declaration` function:

```
(lps:set-mode-declaration predicate mode-template)
```

An error is signalled if *predicate* is not a symbol, if *predicate* is not a defined LogLisp procedure, or if *mode-template* is not a valid mode declaration. In order for the new mode declaration to take effect, *predicate* must now be recompiled.

The function call

```
(lps:show-mode-declaration predicate)
```

returns the current mode declaration for *predicate*. The same restrictions on *predicate* are enforced as with `lps:set-mode-declaration`.

Chapter 5

The LogLisp User Interface

The LogLisp user interface includes extensions to both the Vax Lisp environment and an Emacs environment along with code which makes it possible for the two environments to cooperate with each other much as their counterparts do in the Symbolics Lisp machine environment.

On the Vax Lisp end, the extensions consist of

1. an "input editor" which allows the programmer to perform Emacs-like operations on the command or data that is currently being entered,
2. a "command processor" which allows the programmer to invoke LogLisp features as commands rather than by calling them as Lisp functions, and
3. an extended function definition facility to enhance function documentation, debugging and editing.

The first two extensions are much like (in fact are modeled after) similar features in the Symbolics ZetaLisp environment. The third extension came about during several years of Lisp program development at Honeywell and has been found to be generally useful.

The Emacs extensions comprise a set of commands and key bindings tailored specifically for the editing of Lisp code. These include ways to evaluate or compile all or part of a source file into the Lisp environment.

5.1 VaxLisp Extensions

5.1.1 Input Editor

The input editor allows the programmer to perform Emacs-like editing operations on the input (which could be more than one line) in the LogLisp environment. Most of these operations are invoked by the same keystrokes and perform the same operations as their Emacs counterparts, facilitating the user's perception of an integrated system.

A history of the user's top level commands is maintained and a mechanism provided to invoke earlier commands. The commands are retrieved and displayed on the screen (by using the history manipulating commands Esc-P, Esc-N, and Esc-Esc) in such a way that the user is given a chance to modify the command before it is executed.

A complete description of the key bindings available in the input editor follows. The up-arrow (\uparrow) symbol stands for the control key, while Esc stands for the escape key. This information is available online by typing \uparrow H.

Null Set-Mark-Command. Sets the mark at the current position of the dot.

\uparrow A Beginning-Of-Line-Command. Moves the dot to the beginning of the current line.

\uparrow B Backward-Character-Command. Moves the Dot backward one character. No action is taken if Dot is already at the beginning of the buffer.

\uparrow C Return-To-Top-Level-Command. Terminate any pending input, and return to Lisp top level

\uparrow D Delete-Next-Character-Command. Deletes the character immediately following the Dot. If the Dot is at the end of the buffer, then no action is taken.

\uparrow E End-Of-Line-Or-Enter-Editor-Command. If the input buffer is empty, enter the editor, otherwise move to end of line.

\uparrow F Forward-Character-Command. Moves the Dot forward one character. No action is taken if Dot is already at the end of the buffer.

BELL Illegal-Operation-Command. Signals the user that an illegal operation was attempted. No message is printed.

Backspace Describe-Bindings-Command. Usually prints a list of the current key bindings, with a numeric argument also gives documentation for each. During command entry, prints a list of acceptable inputs.

LINEFEED Newline-Command. Inserts a newline character into the buffer.

↑K Kill-To-End-Of-Line-Command. Characters following the Dot on the current line are killed.

***L** Redisplay-Command. Clears the terminal display, then refreshes.

Return Newline-Command. Inserts a newline character into the buffer.

↑N Next-Line-Command. Moves dot to *PREFIX-ARGUMENT*'th next line, attempting to maintain same column position.

↑P Previous-Line-Command. Moves dot to *PREFIX-ARGUMENT*'th previous line, attempting to maintain same column position.

↑T Twiddle-Characters-Command. Transposes the two characters around the Dot. If at the end of a line, then the preceding two characters are twiddled.

↑U Argument-Prefix-Command. Provides the prefix argument to the next command. If followed by digits, will repeat the number of times specified by treating the digits as a decimal number.

↑W Wipe-Region-Command. Kills the characters between the Dot and the Mark. I.e. they are deleted from the input buffer, and placed in the kill buffer.

↑X-↑X Exchange-Dot-And-Mark-Command. Exchanges the Dot and the Mark

↑Y Yank-Command Inserts the contents of the kill buffer before the dot, the mark is set to the previous location of the dot. Most commands that delete more than a single character, place the text that they delete in the kill buffer.

Esc-↑B Backward-Sexp-Command Moves the Dot to the beginning of the sexp immediately preceding the Dot if the Dot is between expressions, or the beginning of the current sexp if the Dot is inside an sexp.

Esc-^{*}F Forward-Sexp-Command Moves the Dot to the end of the sexp immediatly following the Dot if the Dot is between expressions, or the end of the current sexp if the Dot is inside an sexp.

Esc-^{*}K Kill-Next-Sexp-Command Kills the sexp following the dot.

Esc-[↑]P Toggle-Pretty-Print-Command. Toggles the setting of ***PRINT-PRETTY***. With numeric argument sets ***PRINT-PRETTY*** to T.

Esc-Esc List-History-List-Command. Lists the contents of the history list.

Esc-Space Complete-From-History-List-Command. Attempt to find an entry on the history list that matches the input typed so far, and replace the entire contents of the input buffer with the matching entry. If there is no matching entry, an input editor error is signalled.

Esc-(Backward-Paren-Command. Move backward to matching open parenthesis. With numeric argument, moves over prefix-argument number of open parentheses.

Esc-) Forward-Paren-Command. Move forward to matching close parenthesis. With numeric argument, moves over prefix-argument number of close parentheses.

Esc-. Edit-Definition-Command. Insert an EDIT-DEFINITION form into the input buffer. See section 5.2.

Esc-< Beginning-Of-Buffer-Command. Moves the dot to the beginning of the buffer.

Esc-> End-Of-Buffer-Command. Moves the dot to the end of the buffer.

Esc-B Backward-Word-Command. Moves the Dot to the beginning of the word immediatly preceeding the Dot if the Dot is between words, or the beginning of the current word if the Dot is inside a word. Word characters are defined by the current ***SYNTAX-TABLE***.

Esc-D Delete-Next-Word-Command. Deletes the next word in the buffer. Word characters are defined by the current ***SYNTAX-TABLE***

Esc-F Forward-Word-Command. Moves the Dot to the end of the word immediatly following the Dot if the Dot is between words, or the end of the current word if the Dot is inside a word. Word characters are defined by the current ***SYNTAX-TABLE***.

Esc-N Get-Next-Input-Command. Replace the current input with the next input in the command input queue. You can repeat this to get up to *MAX-HISTORY-LIST-SIZE* commands. Prefix arguments can be used to skip intervening command lines.

Esc-P Get-Last-Input-Command Replace the current input with the previous input in the command input queue. You can repeat this to get up to *MAX-HISTORY-LIST-SIZE* commands. Prefix arguments can be used to skip intervening command lines.

Esc-b Backward-Word-Command. Moves the Dot to the beginning of the word immediately preceding the Dot if the Dot is between words, or the beginning of the current word if the Dot is inside a word. Word characters are defined by the current *SYNTAX-TABLE*.

Esc-d Delete-Next-Word-Command. Deletes the next word in the buffer. Word characters are defined by the current *SYNTAX-TABLE*.

Esc-f Forward-Word-Command. Moves the Dot to the end of the word immediately following the Dot if the Dot is between words, or the end of the current word if the Dot is inside a word. Word characters are defined by the current *SYNTAX-TABLE*.

Esc-n Get-Next-Input-Command. Replace the current input with the next input in the command input queue. You can repeat this to get up to *MAX-HISTORY-LIST-SIZE* commands. Prefix arguments can be used to skip intervening command lines.

Esc-p Get-Last-Input-Command Replace the current input with the previous input in the command input queue. You can repeat this to get up to *MAX-HISTORY-LIST-SIZE* commands. Prefix arguments can be used to skip intervening command lines.

Esc-Delete Delete-Previous-Word-Command. Deletes the word preceding the Dot in the buffer. Word characters are defined by the current *SYNTAX-TABLE*.

^_ Spawn-Dcl-Command. Creates a new DCL process and attaches the user's terminal to it. Logout is used to return. Current command line is lost.

Space.. Self-Insert-Command. Inserts *LAST-KEY-STRUCK* immediately before the Dot.

) Paren-Flash-Command. Inserts a close parenthesis, then flashes the cursor on the matching open parenthesis.

*../ Self-Insert-Command. Inserts *LAST-KEY-STRUCK* immediately before the Dot.

0..9 Digit. If specifying a prefix argument, determines the prefix argument, otherwise inserts itself into the buffer before the Dot.

:- Self-Insert-Command. Inserts *LAST-KEY-STRUCK* immediately before the Dot.

Delete Delete-Previous-Character-Command Deletes the character immediately preceding the Dot. If Dot is at the beginning of the buffer, then no action is taken.

5.1.2 The LogLisp Command Processor

In most Lisp environments the user interacts with the system by typing Lisp forms to be evaluated by the interpreter. The result of the evaluation is then printed back to the user. Naturally, since the input is Lisp, there are a number of parentheses in the input, which some users find a nuisance. We have extended the input processing of LogLisp to support a command oriented interaction. That is, the user may invoke any of a number of LogLisp functions by simply typing the function's name. Arguments (if any) are prompted for and entered in a similar manner. Full input editing (as described previously) is supported on the commands, and help is available during the command by typing `↑H` (or backspace). The forms of these commands are listed here, along with the pages in this manual where they are described.

Command	Page
<i>all goal* keyword-option-pairs*</i>	13
<i>any k goal* keyword-option-pairs*</i>	15
<i>assert-clause clause</i>	16
<i>compile-predicate predicate-specifier</i>	16
<i>delete-predicate predicate-specifier</i>	18
<i>get-predicate predicate</i>	29
<i>get-predicate-clauses predicate</i>	30
<i>list-all-clauses</i>	30
<i>list-all-predicates</i>	31
<i>one goal* keyword-option-pairs*</i>	31
<i>pprint-predicate predicate-specifier &optional stream</i>	31
<i>query goal-list keyword-option-pairs*</i>	31
<i>retract-clause clause</i>	39
<i>set-mode-declaration predicate mode-template</i>	39
<i>show-mode-declaration predicate</i>	43
<i>trace-logic [:all predicate*]</i>	44
<i>uncompile-predicate predicate-specifier</i>	47
<i>untrace-logic [:all predicate*]</i>	47

In addition to the LogLisp commands, commands for several Lisp functions are available. As with the LogLisp commands, prompting for arguments is done, and help is available by typing *^H*. These commands follow:

Select-Editor Choose among the set of allowable editors listed in the variable *input-editor:*allowable-editors**. The currently allowable editors are Gnu emacs, and the builtin VAXLISP editor. Both editors are capable of doing Edit-Definition, and evaluation, however most of the advanced editing functions such as *select-system-as-tag-table*, are only supported in Gnu.

Show-Editor Prints the name of the currently selected editor.

Set-Package Change the current package to the one named.

Show-Package Print the name of the current package.

Use-Package Use the named package.

Load-File Load the named file. Keyword arguments available are *verbose* (which defaults to the value of **load-verbose**), *print* (which defaults to *nil*), and *if-does-not-exist* (which can take the values *:error* or *:ignore* and defaults to *:error*).

Load-System Load the named system. Keyword arguments available are *condition* (which can take on the values *:always*, *:never*, or *:newly-compiled* and defaults to *:newly-compiled*), *query* (which defaults to *t*), *silent* (which defaults to *nil*), and *simulate* (which defaults to *nil*).

Compile-System Compile the named system. Keyword arguments available are *condition* (which can take on the values *:always*, *:new-source*, or *:never* and defaults to *:new-source*), *query* (which defaults to *t*), *silent* (which defaults to *nil*), *simulate* (which defaults to *nil*), and *load* (which can take on the values *:everything*, *:newly-compiled*, or *:nothing* and defaults to *:newly-compiled*).

Compile-File Compile the named file. Keyword arguments available are *listing* (which defaults to *nil*), *machine-code* (which defaults to *nil*), *output-file*, *verbose* (which defaults to the value of **compile-verbose**), *warnings* (which defaults to the value of **compile-warnings**), *optimize-speed*, *optimize-space*, *optimize-safety*, and *optimize-cspeed* (each of the latter four can take on integer values from 0 to 3 and defaults to 1).

Apropos Prompts for a string and a package and finds and prints all symbols in that package which contain the string.

Trace Turn tracing on for named functions. Keyword arguments available are *debug-if*, *pre-debug-if*, and *post-debug-if*, *during*, *print*, *pre-print*, *post-print*, *step-if*, and *suppress-if* (all of which default to the value of *com-trace-ignore*).

Untrace Turn tracing off for named functions.

5.1.3 Extended Function Documentation Facility—*defun+*

We have extended the semantics of Common Lisp's *defun* macro to allow (1) more complete function documentation, (2) better argument type checking for debugging purposes, and (3) simplified function editing through Emacs. This does not mean that we have redefined *defun*; rather, we have made these features available through another macro called *defun+*. This macro allows the programmer to document individual aspects of a function such as arguments, return values and side effects in such a way that the documentation can be extracted easily by other programs. It works by parsing a *defun+*

form, extracting documentation information and building a standard defun form to be evaluated. The documentation information is packaged into a structure and placed on the property list of the symbol whose function is being defined. We have extended the Common Lisp documentation function to access this structure and pretty-print a string with this information. The syntax of `defun+` is:

```

(defun+ name
  ( {var [doc-string] |
    (var [:TYPE type-spec] [doc-string]))*
    [&OPTIONAL
      {var [doc-string] |
        (var [initform [svar]] [:TYPE type-spec]
          [doc-string]))*]
    [&REST var [doc-string]]
    [&KEY
      {var [doc-string] |
        ({var | (keyword var))
          [initform [svar]] [:TYPE type-spec]
            [doc-string]))*]
    [&AUX {var | (var [initform [svar]]
      [:TYPE type-spec]))*])
  [:DESCRIPTION doc-string]
  [:SIDE-EFFECTS doc-string]
  [:RETURNS ({type-spec [doc-string]})]
  [:INLINE {t | nil}]
  [:HISTORY ({"date name doc"})]
  {declaration}*
  {form}*)

```

As an example, consider the following `defun+` form:

```

(defun+ foo
  ((n :type fixnum "This arg better be a fixnum")
   (some-cons :type cons "And this a cons"))
  :description "This is a mysterious function."
  :side-effects "Plenty."
  :returns
    (simple-string
      "This declares the first return value type"
      hash-table
      "This declares the second")
  :inline nil
  :history ("03-29-82 Jack Hack Created"
    "04-30-86 Jill Hose Modified")

```

```
<function body>
```

```
.)
```

All of the documentation strings in this definition will be stored and accessible via the extended documentation function. In addition, if this form is loaded into Lisp from a file, the name of the file will be stored on the property list of `foo`, making it possible for the `Lisp:edit-definition` function to retrieve the file name and hand it to Emacs when the user requests that `foo` be edited. See section 5.2.

Finally, note that the `foo` function example declares the first argument to be a fixnum and the second a cons. This will cause the appropriate declarations to be inserted before the body of the function in the resulting `defun` form. However, VAX Lisp does not check the types of these arguments at function invocation time even though they have been declared. Therefore, `defun+` puts explicit type checking code in the `defun` for all arguments. You can turn off this checking only for compiled code, and only for functions compiled with the `compile-file` function. This action is controlled by the global variable `*include-type-checks-in-compiled-defun+-code*`. If you do not wish to have type checking performed for a group of functions, you must set this variable to `T`, then compile the files that include the functions, and load the compiled files. The rationale is that normally you will want type checking performed until you have debugged a system, at which time would recompile the whole system with optimizations on, and this variable bound to `T`.

5.2 Emacs Extensions

The particular Emacs editor we have chosen for LogLisp program development is GNU Emacs. GNU runs on a large number of different computer systems, and may be freely distributed. GNU is a very powerful and well documented editor with a built-in mode for editing Lisp programs (see the *GNU Emacs Manual*). We provide a set of extensions to the standard Lisp mode which further facilitate editing Lisp code within Emacs. The following is a list of the commands making up these extensions, along with their key bindings, if any, and a short description of the use and importance of each.

`comment-out-region` (binding: `^C-;`) Comments out the current region, that is, puts semi-colons in the first column of each line in region. With an argument, takes the semi-colons out.

compile-buffer (binding: ↑C ↑B)

Compile the current buffer, and then load it into LogLisp.

compile-defun (binding: ↑C ↑D)

Compile the current top level form, and then load it into LogLisp.

compile-region (binding: ↑C ↑R)

Compile the current region, and then load it into LogLisp.

edit-definition (unbound).

Edit the definition for a given symbol. When the user asks to edit the definition of a symbol, Emacs will temporarily pause back to Lisp (which must be running, and have the definition loaded using `defun+`), to get the source file information. Once the source file information has been retrieved, Emacs is automatically restarted, and the file containing the definition is visited, and the definition located and displayed.

electric-lisp-semi (binding: ;)

Inserts the appropriate number of comment characters (semi-colons) given the current indentation. With a numeric argument, inserts that many semi-colons. Correctly handles semi-colons within strings, comments and wrapped lines.

evaluate-buffer (binding: ↑C B)

Evaluate the current buffer.

evaluate-defun (binding: ↑C D)

Evaluate the current top level form.

evaluate-region (binding: ↑C R)

Evaluate the current region.

find-unbalanced-parens (unbound)

Finds any unbalanced parentheses in the current buffer.

lisp-indent-line (binding: TAB or ↑I)

Indent current line appropriately.

reenter-after-eval Variable

If true, emacs will be reentered after doing evaluates. If specified as the symbol `:silently`, emacs will assume that the screen did not change and as a result will not redisplay the screen. This can make evaluation

faster, but can occasionally cause the screen to appear incorrect. The default value is nil.

setup-defun+documentation (binding: Esc--)

Inserts a defun+ form template into the file. See section 5.1.3.

update-attribute-list (unbound)

Inserts default values for the file attributes **Mode**, **Package**, **Base**, and **Syntax**. The file attribute list appears at the beginning of a file and is delineated by the characters "--". To keep Lisp from getting confused when it loads the file, the line appears as a comment. A typical attribute list is:

```
;;; -- Package: User; Mode: Lisp; Base: 10;  
          Syntax: Common-Lisp; --
```

Emacs uses this line to determine what package in which to put forms when they are VaxLisp evaluated. If the package specified in the attribute list differs from the package specified by an **in-package** form near the top of the file, then forms evaluated from Emacs will be in different packages than the same forms loaded directly from a file. **update-attribute-list** makes sure that these package specifications are the same. Gnu checks to make sure that the file's attribute list is consistent with any **in-package** forms appearing in the file whenever the file is read into emacs.

Complete online documentation is provided for all Gnu extensions and is accessible via the Gnu help command **↑H**. Good places to start are with the key bindings (**↑H B**), and lisp mode description (**↑H M**). Both of these commands should be executed from within Gnu after visiting a ".lisp" file.

Index

- *LAST-KEY-STRUCK* 54
- *load-verbose* 55
- :ignore-duplicates keyword to `lps:query` 16
- :index keyword example 29
- :index keyword to `lps:defpredicate` 14
- :mode keyword to `lps:defpredicate` 46
- :solution-limit keyword to `lps:query` 16
- :template keyword to `lps:query` 15
- all macro 18, 55
- any macro 18, 55
- applicative forms, reduction semantics for 32
- Apropos 56
- argument to predicate 3
- argument to predicate, syntax 13
- argument type checking 56
- Argument-Prefix-Command 51
- assert-clause function 13, 43, 55
- atom 9
- atoms, reduction semantics for 32
- backtracking 7
- backtracking, limited 7
- backward chaining 4
- Backward-Character-Command 50
- Backward-Paren-Command 52
- Backward-Sexp-Command 51
- Backward-Word-Command 52, 53
- Beginning-Of-Buffer-Command 52
- Beginning-Of-Line-Command 50
- breadth-first search 7
- call special resolution form 25
- clause 4
- clause goal syntax 13
- clause head syntax 13
- clause syntax 13
- clause syntax error handling 13
- command history 50
- command processor 12, 49
- comment-out-region Emacs Lisp mode function 59
- Common Lisp 1, 2, 34
- Common Lisp special forms, reduction semantics for 34
- compile-buffer Emacs Lisp mode function 60
- compile-defun Emacs Lisp mode function 60
- Compile-File 56
- compile-predicate function 43, 55
- compile-region Emacs Lisp mode function 60
- Compile-System 56
- compiling LogLisp 43
- compiling LogLisp files 44

- Complete-From-History-List-Command 52
- context switching 7
- controlling search 19
- deduction tree 3, 7
- defpredicate macro 14, 43
- defun+ 56
- defun- macro example 58
- defun+ macro syntax 57
- Delete-Next-Character-Command 50
- Delete-Next-Word-Command 52, 53
- delete-predicate function 30, 55
- Delete-Previous-Character-Command 54
- Delete-Previous-Word-Command 53
- depth-first runaway 21
- depth-first search 7
- dereferencing 10
- Describe-Bindings-Command 51
- Digit 54
- Digital Command Language 12
- documentation function, extended 57
- edit-definition Emacs Lisp mode function 60
- Edit-Definition-Command 52
- editing functions 56
- editing Lisp code 59
- electric-lisp-semi Emacs Lisp mode function 60
- Emacs editor 0, 2, 49
- emacs editor 29
- Emacs extensions 59
- End-Of-Buffer-Command 52
- End-Of-Line-Or-Enter-Editor-Command 50
- environment 6, 7
- eval. Lisp function 8
- evaluate-buffer Emacs Lisp mode function 60
- evaluate-defun Emacs Lisp mode function 60
- evaluate-region Emacs Lisp mode function 60
- evaluation 8
- Exchange-Dot-And-Mark-Command 51
- fail example 20, 21
- fail special form 20
- failing a goal 20
- file attribute list 61
- find-unbalanced-parens Emacs Lisp mode function 60
- Forward-Character-Command 50
- Forward-Paren-Command 52
- Forward-Sexp-Command 52
- Forward-Word-Command 52, 53
- function documentation 56
- Get-Last-Input-Command 53
- Get-Next-Input-Command 53
- get-predicate function 30, 55
- get-predicate-clauses function 31, 55
- Gnu Emacs editor 1
- goal 4
- goal-list argument to `lps:query` function 15
- ground special form 38
- Honeywell Systems and Research Center 0
- Horn clause 1
- if, reduction semantics for 35
- Illegal-Operation-Command 50

- index path specifier 27
- indexing clauses 26
- indexing on structured arguments 28
- inference engine 6
- input editor 49
- Input Editor 50
- input line editor 12, 13
- input query 4
- input-editor:*allowable-editors* 55
- instantiation of logic variables 5
-
- Key bindings 50
- Kill-Next-Sexp-Command 52
- Kill-To-End-Of-Line-Command 51
- knowledge base 4, 13
-
- Lisp Commands 55
- lisp-indent-line Emacs Lisp mode function 60
- list-all-clauses function 31
- list-all-predicates function 31
- List-History-List-Command 52
- Load-File 55
- Load-System 56
- loading LogLisp 11
- loading the LPS on a VAX 12
- logic constant 9
- Logic language 1
- Logic language. nested calls 2
- logic procedure 4
- logic procedure calling 5
- logic program 3
- logic programming, object 4
- logic special form 37
- logic variable 4, 9
- logic-and example 24
- logic-and special resolution form 24
- logic-ground special form 38
- logic-if special resolution form 25
- logic-or example 25
- logic-or special resolution form 25
- Loglisp/Emacs interface 14
- LPS Compiler System/Subsystem Specification 0
- LPS Functional Description 0
- LPS Interpreter System/Subsystem Specification 0
- LPS Test Plan 0
- lps:all macro 18, 55
- lps:all macro, examples 18
- lps:any macro 18, 55
- lps:any macro, examples 18
- lps:assert-clause function 13, 43, 55
- lps:call special resolution form 25
- lps:compile-predicate function 43, 55
- lps:defpredicate example 14
- lps:defpredicate macro 14, 29, 43
- lps:defpredicate syntax error handling 14
- lps:delete-predicate function 30, 55
- lps:fail example 20, 21
- lps:fail special form 20
- lps:get-predicate function 30, 55
- lps:get-predicate-clauses function 31, 55
- lps:ground special form 38
- lps:list-all-clauses function 31
- lps:list-all-predicates function 31
- lps:logic special form 37
- lps:logic-and example 24
- lps:logic-and special resolution form 24
- lps:logic-ground special form 38
- lps:logic-if special resolution form 25
- lps:logic-or example 25

- lps:logic-or special resolution form 25
- lps:one macro 18, 55
- lps:one macro. examples 18
- lps:pprint-predicate function 30, 55
- lps:quasi-quote special form 37
- lps:query function 15, 55
- lps:quit example 19
- lps:quit special form 19
- lps:reduce-term form 31
- lps:reduce-term reader macro 32
- lps:retract-clause function 29, 55
- lps:set-mode-declaration function 47, 55
- lps:show-mode-declaration function 47, 55
- lps:succeed example 20
- lps:succeed special form 20
- lps:suspend special form 22
- lps:trace-logic macro 39, 55
- lps:uncompile-predicate function 44, 55
- lps:untrace-logic macro 39, 55
- lps:variable special form 38

- macros. reduction semantics for 33
- mode declaration semantics 45
- mode declaration, structured 45
- mode declaration, unstructured 45
- mode declarations. application rules 46
- mode template 45

- Newline-Command 51
- Next-Line-Command 51
- node stack 26
- node stack growing 26
- node stack overflow 26
- node suspension 7

- one macro 18, 55

- Paren-Flash-Command 54
- pprint-predicate function 30, 55
- predicate 3
- predicate calculus 1, 3, 4
- predication 3
- Previous-Line-Command 51
- progn, reduction semantics for 35
- Prolog 1, 7

- quasi-quote special form 37
- query examples 16
- query function 15, 55
- quit example 19
- quit special form 19
- quitting a query 19
- quote, reduction semantics for 36

- Recursive call to Logic, example* 23
- recursive calls to Logic 15
- Redisplay-Command 51
- reduce-term form 31
- reduce-term reader macro 32
- reduction 2, 3, 8, 9, 10, 31
- reenter-after-eval Emacs Lisp mode variable 60
- resolution 1, 7
- retract-clause function 29, 55
- Return-To-Top-Level-Command 50
- Rome Air Development Center 0

- Select-Editor 55
- Self-Insert-Command 53, 54
- Set-Mark-Command 50
- set-mode-declaration function 47, 55
- Set-Package 55
- setq, reduction semantics for 36

- setup-defun--documentation Emacs
 - Lisp mode function 61
- Show-Editor 55
- show-mode-declaration function 47.
 - 55
- Show-Package 55
- solution cost heuristic 7, 22
- Spawn-Dcl-Command 53
- special resolution forms 24
- succeed example 20
- succeed special form 20
- succeeding a goal 20
- Sun machines 1
- suspend special form 22
- suspending a node 22
- Symbolics Environment 49
- Symbolics machines 1
-
- The LogLisp Command Processor
 - 54
- The LogLisp User Interface 49
- Toggle-Pretty-Print-Command 52
- Trace 56
- trace-logic macro 39, 55
- Twiddle-Characters-Command 51
-
- uncompile-predicate function 44, 55
- unification 3, 5
- Untrace 56
- untrace-logic macro 39, 55
- update-attribute-list Emacs Lisp mode
 - function 61
- Use-Package 55
- user interface 2
- using the lps package 12
-
- variable special form 38
- VAX machines 1, 2, 11
-
- waiting heap 7, 22
-
- waiting node 22
- Warren Prolog machine 43
- Wipe-Region-Command 51
-
- Yank-Command 51

APPENDIX A

LogLisp Programming System Computer Operations Manual

**F30602-84-C-0121
In Fulfillment of CDRL B007**

**J. Carciofini T. Colburn
G. Hadden**

**Honeywell Systems And Research Center
Mail Station MN65-2100
P.O. Box 1361
Minneapolis MN 55440
Phone (612) 782-7306**

June 28, 1987

Contents

1 General	2
1.1 Purpose of the Computer Operations Manual	2
1.2 Project References	2
1.3 Terms and Abbreviations	3
2 System Overview	4
2.1 System Application	4
2.2 System Organization	4
3 Installation	6
3.1 Prerequisites	6
3.2 Detailed Steps	7
3.2.1 Restore the Lisp Extensions from Tape	7
3.2.2 Restore GNU Emacs from Tape	8
3.2.3 Install the LPS	8
4 Recompiling	10
4.1 Recompiling the LISPEXTEN	10

Chapter 1

General

1.1 Purpose of the Computer Operations Manual

The objective of this Computer Operation Manual for the LogLisp Programming System (LPS) RADC Contract F30602-84-C-0121, is to provide computer control and computer operator personnel with a detailed description of the operations and command sequences required to bring up, from source files, a working instance of the LPS under the VAX/VMS operating system.

1.2 Project References

Documents applicable to the history and development of the LogLisp Programming System project are:

1. Carciofini, J., Colburn, T. and Hadden, G., "LogLisp Programming System Users Manual", Honeywell Systems and Research Center, April 1, 1987.
2. Beane, J., Carciofini, J. and Colburn, T., "LogLisp Programming System Functional Description", Honeywell Systems and Research Center, April 12, 1985.
3. Beane, J., Carciofini, J., Colburn, T. and Lukat, R., "LogLisp Programming System Test Plan", Honeywell Systems and Research Center, August 1, 1985.

CHAPTER 1. GENERAL

4. Carciofini, J., Colburn, T. and Lukat, R., "LogLisp Programming System Interpreter System Subsystem Specification", Honeywell Systems and Research Center, December 16, 1985.
5. Carciofini, J., Colburn, T. and Lukat, R., "LogLisp Programming System Compiler System Subsystem Specification", Honeywell Systems and Research Center, April 15, 1986.

1.3 Terms and Abbreviations

Technical terms, abbreviations and acronyms unique to this project are defined in this report where they are introduced.

Chapter 2

System Overview

2.1 System Application

The LogLisp Programming System (LPS) is a total Lisp programming environment extended to provide logic programming capabilities. The general nature of programs developed with the LogLisp Programming System can be classified as artificial intelligence applications.

2.2 System Organization

The LPS implementation consists of three parts. The three parts are:

1. VAXLISP Version 2.1 - This is Digital Equipment Corporation's implementation of Common Lisp for VAX/VMS.
2. LISP Extensions - The LISP Extensions consist of several extensions to the VAXLISP programming environment. A list of the major facilities provided follows:
 - The LogLisp interpreter and compiler.
 - A facility for maintaining a collection of lisp source files.
 - An input editor so that commands lines may be edited using Emacs-like key bindings.
 - A command processor that interprets commands as well as evaluating Lisp expressions.
 - Support for communication with GNU Emacs.

CHAPTER 2. SYSTEM OVERVIEW

- An extended documentation facility.
3. GNU Emacs - GNU Emacs is a free editor developed by the Free Software Foundation. GNU Emacs is not in the public domain: it is copyrighted, but the restrictions of the copyright are intended to keep GNU Emacs free and prevent commercialization of any part of the software. The GNU Emacs we are distributing contains several extensions we have developed to enhance the LPS development environment. These extensions are written in the GNU Emacs extension language. These extension provide aids for the creation/editing of Common Lisp/LPS code.

This document assumes that VAXLISP Version 2.1 has been acquired from Digital Equipment Corporation and installed according to the instructions received with the installation kit. If you have not installed VAXLISP you will need to install it before you are able to run the LPS.

Chapter 3

Installation

This chapter describes the steps necessary to load the LPS from tape and to install the LPS on a VAX/VMS system. First the prerequisites for the installation are presented, then the installation procedure itself.

3.1 Prerequisites

The prerequisites for installing the LPS on a VAX/VMS system are listed in this section. Please be sure that all the prerequisites have been satisfied before attempting to install the LPS.

First you must be running VMS version 4.4 and have VAXLISP version 2.1 installed on your system.

You must have the LPS installation kit. This kit consists of this document, the LPS distribution tape and the GNU distribution tape.

You will also need 70000 blocks of disk space. This space does not have to be on one disk. The system can be divided across disks as follows:

- 40000 blocks for the Lisp Extensions.
- 30000 blocks for GNU Emacs.

In summary, the prerequisites that must be met before attempting to install the LPS are:

- VMS version 4.4
- VAXLISP version 2.1
- LPS installation kit.

CHAPTER 3. INSTALLATION

- 40000 blocks of disk space for the Lisp Extensions.
- 30000 blocks of disk space for GNU Emacs.

3.2 Detailed Steps

This section provides an overview of the installation procedure. Each of the high level steps described here will be expanded in detail in the following sections. Assuming the prerequisites have been met the steps for installing the LPS are:

- Restore the Lisp Extensions from tape.
- Restore GNU Emacs from tape.
- Install the Lisp Extensions and GNU Emacs.

3.2.1 Restore the Lisp Extensions from Tape

You must first decide where the 40000 blocks of Lisp Extensions will reside on your system. For the rest of this installation procedure we will assume that the Lisp Extensions directory tree is rooted at `LISPEXTEN_DISK:[LISPEXTEN]`. The Lisp Extensions distribution tape is written in VMS backup format at 1600bpi. Now load the Lisp Extensions tape on an appropriate tape drive. Then enter the following commands to restore the tape contents, substituting your tape device name for `TAPE:` and the appropriate pathnames for `LISPEXTEN_DISK:[LISPEXTEN]`. It is not necessary that these files be owned by the user `SYSTEM`; they can actually be owned by anyone at your site.

```
$ allocate TAPE:
$ mount TAPE:/foreign
$ backup TAPE:LISPEXTEN.bck/select=[LISPEXTEN...]*.*.* -
    LISPEXTEN_DISK:[LISPEXTEN...]*.*.* -
    /verify/owner=system
$ dismount TAPE:
$ deallocate TAPE:
```

You can now remove the tape from the tape drive. The Lisp Extensions code has now been restored from the tape to your system.

CHAPTER 3. INSTALLATION

3.2.2 Restore GNU Emacs from Tape

You must now decide where the 30000 blocks of GNU Emacs will reside on your system. For the rest of this installation procedure we will assume that the GNU Emacs directory tree is rooted at `GNU_DISK:[GNU]`. The Lisp Extensions distribution tape is written in VMS backup format at 1600bpi. Now load the GNU Emacs tape on an appropriate tape drive. Then enter the following commands to restore the tape contents, substituting your tape device name for `TAPE:` and the appropriate pathnames for `GNU_DISK:[GNU]`. It is not necessary that these files be owned by the user `SYSTEM`; they can be owned by anyone at your site.

```
$ allocate TAPE:
$ mount TAPE:/foreign
$ backup TAPE:GNU.bck/select=[GNU...]*.*.* -
      GNU_DISK:[GNU...]*.*.*             -
      /verify/owner=system
$ dismount TAPE:
$ deallocate TAPE:
```

You can now remove the tape from the tape drive. The GNU Emacs code has now been restored from the tape to your system.

3.2.3 Install the LPS

The logical name `LISPEXTEN` must be defined to be rooted at `LISPEXTEN_DISK:[LISPEXTEN...]*.*.*`. Modify the system startup file, `SYSS$MANAGER:SYSTARTUP.COM`, to contain the following lines.

```
$! LogLisp Programming System (LPS) global definitions
$ define lispexten LISPEXTEN_DISK:[LISPEXTEN.] -
      /translation_attributes=concealed/system
```

You should also type this command to VMS now, so that it takes effect. Now we need to define two symbols in the system login. Add the following two lines to your `SYSS$MANAGER:SYSTARTUP.COM` file.

```
$ lispexten_init := @LISPEXTEN:[BUILD.VMS]lispexten-init
$ gnu_emacs_init := @GNU_DISK:[GNU.EMACS]emacs
```

CHAPTER 3. INSTALLATION

The Installation procedure is now complete. To use the LPS system set up you initialization files as documented in the LPS Users Manual.

Chapter 4

Recompiling

This section describes the procedures for compiling the Lisp Extensions. There are several conditions that may require you to recompile. The more common ones are:

- After installing a new release of VMS.
- After installing a new release of VAXLISP.
- After modifying an LPS system source file.

The installation notes for a new version of VMS or VAXLISP should indicate whether or not you need to recompile lisp files or remake any suspended images. The following sections describe the procedures for recompiling the LPS and the LISPEXTEN.

4.1 Recompiling the LISPEXTEN

We have provided a DCL command procedure that will start a batch job to compile the Lisp Extensions from sources and create a new suspended image. The command procedure is `LISPEXTEN:[BUILD.VMS]make-all.com`. The first argument to the command procedure specifies whether you want to recompile, compile or load the Lisp Extensions. The options are:

- **recompile** - Compile all sources then load the binaries and create a new suspended image.
- **compile** - Compile only those sources that have changed then load the binaries and create a new suspended image.

CHAPTER 4. RECOMPILING

- **load** - Just load the existing binaries and create a new suspended image.

Any legal parameters to the VMS **SUBMIT** command can be used when invoking this command procedure. For example, to recompile the whole system after 10:00pm today enter the following command.

```
$ @LISPEXTEN:[BUILD.VMS]make-all recompile /after=22:00
```

When the batch job finishes it will send a mail message to the user that executed the command procedure indicating success or failure of the run. The default log file is `LISPEXTEN:[BUILD.VMS]make-all.log`. If the run fails look at this log file to determine the problem. After correcting the problem simply execute the command procedure again.

Running this command procedure may create new versions of binaries and suspended images. To purge these files execute the following command.

```
$ purge LISPEXTEN:[000000...]*.fas,*.sus
```

The suspended image files are quite large (≈ 6000 blocks), so it would be wise to be sure to purge these files with the above command.

MISSION of Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

END

DATE

FILMED

7-88

Dtic